REPORT DOCUMENTATION PAGE

1. Recipient's Reference	2. Originator's Reference	3. Further Reference	4. Security Classification of Document
	AGARD-LS-146	ISBN 92-835-1527-7	UNCLASSIFIED

5. Originator	Advisory Group for Aerospace Research and Development North Atlantic Treaty Organization 7 rue Ancelle, 92200 Neuilly sur Seine, France
6. Title	APPLICATION OF ADA* HIGHER ORDER LANGUAGE TO GUIDANCE AND CONTROL
7. Presented on	20—21 May 1986 in Ottawa, Canada, 2—3 June 1986 in London, United Kingdom, and 5—6 June 1986 in Cologne, Germany.

8. Author(s)/Editor(s)	9. Date
Various	May 1986

10. Author's/Editor's Address	11. Pages
Various	114

12. Distribution Statement	This document is distributed in accordance with AGARD policies and regulations, which are outlined on the Outside Back Covers of all AGARD publications.

13. Keywords/Descriptors

ADA (programming languages)
Computer systems programs
Digital computers
Compilers
Operating systems (computers)

14. Abstract

The need to reduce escalating software life-cycle costs is the raison d'être for Ada. Early experience with the language suggests that the promise of increased software productivity can be fulfilled. However, many problems remain: the need for validated and efficient compilers targeted for computers suitable to the guidance and control application, and software development environments built around Ada, are two among the foremost.

This Lecture Series, sponsored by the Guidance and Control Panel of AGARD, has been implemented by the Consultant and Exchange Programme of AGARD.

* Ada is a registered trademark of the US Government (Ada Joint Program Office)

AGARD-LS-146

AGARD-LS-146

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT , Paris.

7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD LECTURE SERIES No.146

Application of ADA Higher Order Language to Guidance and Control

NORTH ATLANTIC TREATY ORGANIZATION

DISTRIBUTION AND AVAILABILITY ON BACK COVER

AGARD-LS-146

NORTH ATLANTIC TREATY ORGANIZATION

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT

(ORGANISATION DU TRAITE DE L'ATLANTIQUE NORD)

AGARD Lecture Series No.146

APPLICATION OF ADA HIGHER ORDER LANGUAGE TO GUIDANCE AND CONTROL

The material in this publication was assembled to support a Lecture Series under the sponsorship of the Guidance and Control Panel and the Consultant and Exchange Programme of AGARD presented on 20—21 May 1986 in Ottawa, Canada, 2—3 June 1986 in London, United Kingdom, and 5—6 June 1986 in Cologne, Germany.

THE MISSION OF AGARD

The mission of AGARD is to bring together the leading personalities of the NATO nations in the fields of science and technology relating to aerospace for the following purposes:

— Exchanging of scientific and technical information;

— Continuously stimulating advances in the aerospace sciences relevant to strengthening the common defence posture;

— Improving the co-operation among member nations in aerospace research and development;

— Providing scientific and technical advice and assistance to the Military Committee in the field of aerospace research and development (with particular regard to its military application);

— Rendering scientific and technical assistance, as requested, to other NATO bodies and to member nations in connection with research and development problems in the aerospace field;

— Providing assistance to member nations for the purpose of increasing their scientific and technical potential;

— Recommending effective ways for the member nations to use their research and development capabilities for the common benefit of the NATO community.

The highest authority within AGARD is the National Delegates Board consisting of officially appointed senior representatives from each member nation. The mission of AGARD is carried out through the Panels which are composed of experts appointed by the National Delegates, the Consultant and Exchange Programme and the Aerospace Applications Studies Programme. The results of AGARD work are reported to the member nations and the NATO Authorities through the AGARD series of publications of which this is one.

Participation in AGARD activities is by invitation only and is normally limited to citizens of the NATO nations.

The content of this publication has been reproduced directly from material supplied by AGARD or the authors.

Published May 1986

ISBN 92-835-1527-7

SPS

Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

PREFACE

This Lecture Series No.146 on the "Application of Ada® Higher Order Language to Guidance and Control" is sponsored by the Guidance and Control Panel (GCP) of AGARD, and implemented by the Consultant and Exchange Programme of AGARD. The Department of Defense has stated that Ada shall become the single, common computer programming language for Defense mission-critical applications beginning in 1984. The proposal to make Ada a NATO standard High Order Language (HOL) as well, prompted the GCP to provided a Lecture Series embracing the basic structure, theories and principles embodied in this HOL.

The need to reduce escalating software life-cycle costs is the raison d'être for Ada. Early experience with the language suggests that the promise of increased software productivity can be fulfilled. However, many problems remain: the need for validated and efficient compilers targeted for computers suitable to the guidance and control application, and software development environments built around Ada, are two among the foremost.

This Lecture Series addresses both the promise and the problems. Following an introduction to Ada, the structure and features of the language are described. Two special features of Ada, parallel processing and software reusability, are discussed in detail. Compiler development and validation, and the use of Ada in programming environments are described. Finally, the use of Ada in a real-world pilot project is described.

Theodore F.Westermeier
McDonnell Douglas Corporation
St. Louis, Missouri

®Ada is a registered trademark of the US Government (Ada Joint Program Office).

LIST OF SPEAKERS

Lecture Series Director: Dr T.F.Westermeier
129 Lindenwood Avenue
St Charles, MO 63301
USA

SPEAKERS

Mr R.E.Bolz
6751 South Dahlia Court
Modern Programming Languages
Littleton, CO 80122
USA

Dr O.Roubine
Informatique Internationale
Route des Dolines
Les Cardoulines
06560 Valbonne
France

Dr E.Ploedereder
Tartan Laboratories Inc.
477 Melwood Avenue
Pittsburgh, PA 15213
USA

Dr T.P.Baker
Florida State University
Tallahassee, FL 32306
USA

Mr D.Eilers
PPresident
Irvine Compiler Corporation
18021 Sky Park Circle, Suite L
Irvine, CA 92714
USA

Dr M.Selwood
Plessey Defence Systems
Abbey Works
Titchfield
Fareham, Hants PO14 4QA
United Kingdom

CONTENTS

* Paper 2 and Paper 3 have been combined for the purpose of this publication.

INTRODUCTION TO THE APPLICATION OF ADA® TO GUIDANCE AND CONTROL

THEODORE F. WESTERMEIER
MCDONNELL AIRCRAFT COMPANY
MCDONNELL DOUGLAS CORPORATION
P. O. BOX 516
ST. LOUIS, MO 63166

The Department of Defense (DoD) has stated that Ada shall become the single, common computer programming language for defense mission-critical applications beginning in 1984. The proposal to make Ada a NATO standard High Order Language (HOL) as well, prompted the Guidance and Control Panel of AGARD to provide a Lecture Series embracing the basic structure, theories and principles embodied in this HOL.

The need to reduce, or at least contain, escalating software life-cycle costs is the main reason for the existence of Ada. The costs of developing software for the DoD are expected to have a ten-fold growth from an estimated $2.8 billion in 1980 to as much as $30 billion in 1990, as shown in Figure 1. Accordingly, the U.S. Government is sponsoring several initiatives directed toward the development of improved software. These include a program called Software Technology for Adaptable, Reliable Systems, another called the Joint Service Software Engineering Environment, the newly established Software Engineering Institute at Carnegie Mellon University, and the Strategic Computing Program. Organizations outside the U.S. have also embarked on programs for improved software productivity. These include the European Esprit Consortium, and Great Britain's Alvey Programme. A key element in many of these initatives is Ada.

The application of Ada to guidance and control may, at first glance, appear to be a rather narrow focus. (Although Ada was originally intended for embedded computer systems, it nevertheless includes features applicable to a wide application domain. Indeed, there is currently wide-spread interest in Ada in applications that range from business to scientific.) Nevertheless, guidance and control is an important subset of embedded systems used throughout aerospace. If Ada can be successfully applied to guidance and control, it can be used in many embedded systems. Consequently, "guidance and control" can, in many ways, be read as "embedded".

Guidance and control includes a wide range of application programs, as depicted in Figure 2. The list is by no means complete. The trend to closely-interacting and interdependent subsystems results in increasing levels of sophistication. Subsystems that were once outside guidance and control considerations are no longer. For instance, the propulsion system of a vertical take-off and landing aircraft becomes a critical control system effector during the hover mode, supplying control moments in addition to propulsive forces. The increased levels of sophistication dictate the increased use of high performance digital processors and operational software. Consequently, the growth in guidance and control software mirrors the exponential growth of DoD software in general.

Guidance and control systems impose special requirements: the computation times must be acutely limited; the systems are mission critical, and often flight critical; severe limitations are placed on the size, weight, power and cost of the underlying digital mechanizations. These system requirements demand operational software that is efficient, reliable and maintainable.

These special system requirements interact in complex ways and place stringent demands on Ada. For example, the dynamic performance of many guidance and control systems requires that the response times, transport lags, and iteration rates inherent in digital mechanizations be restricted to certain limited values, in order that the system performance not be compromised. This in turn requires that the Ada compiler be efficient. Of course, the higher the throughput of the target computer, the more compiler inefficiency that can be tolerated. But the size, power, weight and cost constraints limit the throughput that can be realized from realistic guidance and control computers. (After all, not many airborne systems can accommodate mainframe computers.)

Early experience with the Ada language suggests that the promise of increased software productivity and reliability can be fulfilled. However, many problems remain: the need for validated and efficient compilers targeted for computers suitable to the guidance and control application, and software development environments built around Ada, are two at the top of the list.

The lectures in this series are intended to emphasize the promise of Ada without minimizing the problems. Accordingly they have been structured to address the following topics:

- o The origin of the "software crisis" and how Ada can alleviate it
- o Deficiencies of other HOL's rectified by Ada
- o Ada language facilities that have special relevance to real-time, embedded systems

®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

- o Prognosis as to Ada's acceptance; some possible roadblocks
- o The role of tasking in embedded applications
- o How Ada supports tasking
- o The productivity implications of reusable software
- o How Ada supports the concept of reusable software
- o Ada compiler development issues: rehostability, optimization, run-time environments, related Ada tools
- o The purpose and mechanics of validating a compiler
- o The economics of compiler development and validation
- o Are all but the most widely-used computers ruled out by the economics?
- o What validation proves, what is doesn't prove
- o Programming with Ada
- o The Ada Programming Support Environment
- o The role of Ada software development tools
- o Practical results and experiences in applying Ada to some real-world, real-time embedded systems.

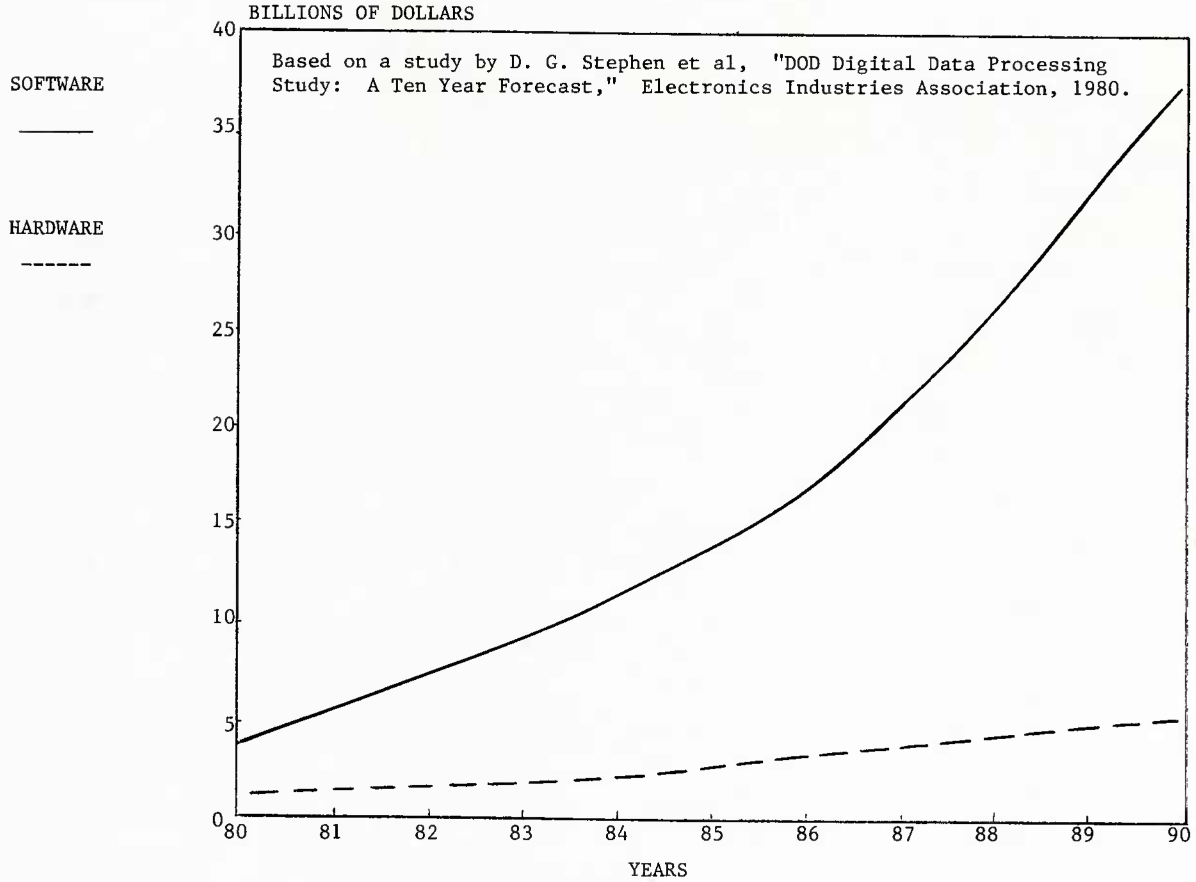
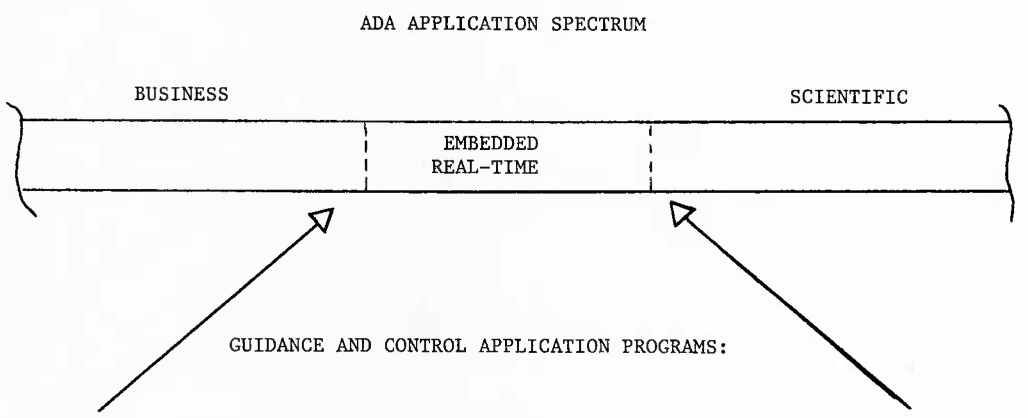


FIGURE 1. DOD Hardware/Software Cost Trends



- DIGITAL FLIGHT CONTROLS
- DIGITAL PROPULSION CONTROLS
- AIDED INERTIAL NAVIGATION
- MULTIFUNCTION SENSORS
- INTEGRATED FLIGHT/WEAPON CONTROL
- ON-BOARD SIMULATION AND TRAINING
- ENERGY MANAGEMENT
- AIR BATTLE MANAGEMENT
- TACTICAL FLIGHT MANAGEMENT

FIGURE 2. The Ada Application Domain

INTRODUCTION TO ADA

by
Richard E. Bolz
6751 S. Dahlia Ct.
Littleton, Co. 80122
USA

SUMMARY

This paper discusses the development of the Ada programming language and current experiences with using the language. It presents the language as an embodiment of good software development principles.

ABSTRACT

The "Software Crisis", first recognized in the late 1960's is exemplified by software that is late, expensive, filled with errors and nearly impossible to maintain. The U.S. Department of Defense (DoD), the largest customer of software in the world decided to sponsor the development of a common high-ordered programming language to give partial response to this continuing crisis.

The language, Ada, has an interesting development history. Public review of the language requirements resulted in a document which is small, concise and readable (counter to what we have come to expect as DoD documentation). The resulting language embodies many of the principles of good software development which have emerged within the past 15 years. These principles include abstraction, information hiding, localization and modularity.

The Ada language enhances transportability by eliminating almost all dependence on an operating system. Hence, such capabilities as exception handling and concurrency become programming language features as opposed to operating system features.

The DoD and NATO have adopted Ada as the common high-order language for all mission critical software (DoD in 1984 and NATO in 1986). The success of the language will not be limited to military systems, however. There appears to be wide-spread interest in the language from such diverse areas as banking, geophysical activity, computer-based training, artificial intelligence and accounting. What started out as a language primarily for embedded computer systems is emerging as a language that can be easily adapted to a variety of application areas.

Experienced users of the language have been quite positive in their evaluations and many managers claim that their software developers have experienced remarkable productivity increases. Also many early results tend to allay fears that generated code and the compilation process would be too slow.

THE SOFTWARE CRISIS

No one in our industry can seriously argue that modern software is trivial to produce. Are there any among us who consistently develop software that comes in on time, on budget, works according to specs and is straightforward to modify? Collectively, these attributes define the software crisis.[1] This phenomenon of difficulty in developing software was first recognized in the mid 1960's, about the time that we began to realize that the problems we were asked to solve were becomming increasingly complex. To compound matters, the emergence of various architectures caused us to desire software which was transportable; we needed (and still need) software which could run (with minor modification) on architectures other than that on which it originally executed.

This "crisis" generated considerable research into ways to do things better. A new appreciation for the system life cycle emerged. Structured X, where X stands for coding programming, analysis, design, testing, walkthrough or any of several other software development activities, started to face us everywhere we turned. Some of these have met with far more success than others. Further, we began to experience an increased interest in developing tools to aid us in various phases of the system life cycle.

THE DEPARTMENT OF DEFENSE RESPONSE

The software crisis (some refer to it as a software depression) was and is most keenly felt by the U.S. Department of Defense (DoD), the largest single user of software in the world. Early in the 1970's the DoD realized that language proliferation was a major cause of software difficulty.[2] It was found that over 450 different languages were used in DoD mission-critical systems. These systems include such diverse application areas as intelligence systems, cryptologic systems related to national security, command and control of military forces, weapons systems and administrative support for such systems.[3] None of the 450 were found to dominate the others as far as frequency of use was concerned (although FORTRAN was used more than any other language).

Mission-critical systems have some characteristics which are rarely found in more common systems such as payroll processing. Parallel processing and real-time response are two such characteristics. Further, in the early 1970's mission critical systems

accounted for over 56% of the total DoD software budget.[4] Some estimates show that software budget exceeding $32 billion annually by 1990 if standardization on a common language is not forthcoming.[5]

In the mid 1970's, the DoD formed the High-Order Language Working Group (HOLWG) to investigate the feasibility of transition to a single language for mission-critical systems. This group was to (1) decide what features should be required in the language, (2) compare existing languages against those requirements and (3) make recommendation to either adopt an existing language or have one developed which meets the requirements.[6]

The DoD expected certain benefits to accrue with a single, high-order language. Among these benefits:[7]

(1) Reduced cost by decreasing (if not eliminating) duplication of language-specific efforts across the lifecycle.
(2) Improved communication by eliminating the "artificial" boundaries which hamper software technology transfer.
(3) Focused research on embedded computer systems software. A plurality of languages makes data gathering very difficult.
(4) Decreased ties to particular vendors who might have the inside track when it comes to supporting their "unique" languages.
(5) Focused energy on the problem at hand rather than on a new language custom built for the project.
(6) Consolidation of fiscal capability resulting in a wider range of software development tools which can provide considerable leverage.
(7) Decreased risk in using existing languages because of their current poor support.

The approach to defining the requirements for a language was somewhat unique. The HOLWG first generated a beginning collection of requirements and called the document STRAWMAN. Of course, this is not a novel approach. Many task forces will meet and draft up a document which they will call "strawman". They will then meet two week later, read their "strawman" and either modify it or accept it as final. The HOLWG did not take this traditional approach. Instead, they submitted STRAWMAN to public review. Reviewers from such diverse communities as academe, science, military, and business took part. The results of this process were folded into the next iteration of the sequence, WOODENMAN. The process continued and produced TINMAN, IRONMAN, revised IRONMAN and finally, STEELMAN, the final definition of the language requirements. Given this "Wizard of OZ" approach it is remarkable that the resulting language was not named Dorothy instead of Ada. Two very pleasant surprises which came from this three-year requirements formulation are that the STEELMAN document is only 22 pages long and that it is quite easy to understand.[8]

A Request For Proposal (RFP) was issued in 1977 for new languages which would meet the emerging requirements. There were 17 responses with most of the proposed languages being based on Pascal. A vigorous competition ensued and the eventual winner was the language submitted by Cll-Honewell Bull, a French corporation. The design team was led by Jean Ichbiah and contained participants from several nations.

The language was officially named "Ada" in the spring of 1979. Prior to that time, the unofficial name of "DoD-1" (it should be obvious why they didn't use the name "DoD-0"). The name Ada honors the mathematician of the 19th century who, as colleague to Charles Babbage, developed an instruction set for the as yet unbuilt analytical engine. She proceeded to use this instruction set to solve some Bernoulli equations. Since no previous algorithm for use on a computing device (real or theoretic) is known, we refer to her as the first computer programmer in history.[9]

THE ENVIRONMENT

> Four of the original goals of the DoD common high-order language effort, which led to the definition of the Ada language, were to:
>
> Address the problem of life-cycle program costs
> Improve program reliability
> Promote the development of portable software, and
> Promote the development of portable software tools
>
> It was recognized from the beginning that these objectives would not be met by the language alone, but by a comprehensive, integrated programming environment.[10]

Early in the development of the requirements of the Ada language, it became apparent that a common language would not be sufficient to address the life-cycle problems of DoD mission-critical software. Support for such a language would be enhanced by having an environment containing certain language-specific tools which could be used by all software developers using Ada. Among these tools would be editors, linker/loaders, debuggers, etc.

In 1978, a workshop was held to attempt to identify what an Ada Programming Support Environment might look like. A document called PEBBLEMAN emerged shortly thereafter and, like STRAWMAN, was submitted to extensive public review. The iterative process terminated with the STONEMAN document in February, 1979. It is important to realize that the STONEMAN document was not to be considered as a collection of firm requirements as was the case with STEELMAN, but as "...a yardstick against which competing versions of an APSE could be measured."[11]

It was apparent that "...there (was) no consensus about what an APSE should look like. The beauty of an APSE is in the eye of the beholder, and there are many beholders... No single APSE could fulfill all the wishes of all the participants."[12] However, there was an agreement on certain characteristics or tools which would most likely be common to all APSEs. This Minimal Ada Program Support Environment (MAPSE) became the focus to attention. It included compilers, link editors, simulators, symbolic (high level language) debuggers, editors, prettyprinters, set-use analyzers, dynamic analysis tools and configuration management routines.

A major consideration for a MAPSE was that tools developed for one environment should be transportable to another environment. Further, tools should be, to the extent possible, interoperable. That is, tools should be able to invoke other tools within the environment. This transportability of tools raises the question of machine dependence. Even though a tool would operate relative to a common language, it still had to execute on a given host architecture. It is clear that transportability of software is hampered if that software is dependent upon its underlying host architecture or operating system.

The writers of the STONEMAN document propose an interesting solution to this transportability delimma: have the MAPSE tools execute on a virtual Ada machine. They propose that a machine <u>dependent</u> Kernal Ada Program Support Environment (KAPSE) provide a machine <u>independent</u> interface to the MAPSE. This KAPSE would contain such items as the APSE data base, database access routines, operator interface routines, operating system interface routines and a complete runtime support package.[13]

THE LANGUAGE

As a result of the public review process, the final set of language requirements (found in the STEELMAN document) included the following language features:

Strong typing and type conversions
Relative and absolute precision
Enumerated and integer types
Variable declaration with initial values
Arrays and records
Sequential, conditional and iterative control structures
Subprograms (functions and procedures)

Encapsulated definitions
Parallel processing
Generic Definition
Exception handling
Representation specification
Interface to other languages

The first group of features can be found, in some form or another, in many algorithmic languages (most notably, Pascal). These features will not be discussed further in this paper. It is the second group of features that makes Ada an interesting language. We will discuss each of them briefly.

ENCAPSULATED DEFINITIONS

This language requirement provides a natural marriage of the software engineering principles of abstraction and information hiding. Abstraction is the ability to deal with things in their real world (problem space) terminology. "The essence of abstraction is to extract essential properties while ommitting inessential details."[14] We need to be able to deal with the functionality of the entity under consideration and defer (for the moment) any consideration of the underlying representation. Information hiding allows us to make inaccessible these detailed decisions of representation thereby forcing the user to rely on the abstractions provided.

The Ada construct for implementing encapsulated definitions is known as the package. A package is a module that allows one to gather together a collection of logically related computational resources. The package has a two part structure: the specification indicates <u>what</u> is available and the body indicates <u>how</u> it is actually accomplished. The specification represents a contract with the user of the package. In fact, the specification indicates all objects and operations which can be exported. The user can make full use of the package by knowing only the specification. The body contains information of interest only to the implementor of the resource. In addition to the implementation details of the operations exported by the package, the body may contain other information which is hidden from all but the implementor. Such information might include utility routines, state memory, file manipulation, etc.

The package concept in Ada was greatly influenced by the concept of a class in SIMULA and the module in MODULA.

PARALLEL PROCESSING

Perhaps the epitome of information hiding is the fact that the Ada language "hides" the very existence of the operating system from the applications programmer. Nowhere is this more apparent than with the Ada model of concurrent execution of communicating

sequential processes. As far as the user is concerned, there is no reliance on the operating system via system calls to effect concurrency. The Ada tasking model allows the user to nicely model the natural concurrency present in many of the problems we are asked to solve. This natural concurrency is especially present in mission-critical systems.

The concurrency model of Ada is based, to a large degree, on the Communicating Sequential Processing (CSP) model of Professor C.A.R. Hoare of Oxford University.[15] In Ada, a task is considered to be executing on its own dedicated virtual processor. It doesn't matter whether the actual execution is on a multicomputer system, a true multiprocessing system or with interleaved execution on a single processor. Further, tasks are assumed to communicate with each other directly (not through an operating system). It should be stated that the actual communication is most likely through operating system calls but that is part of the service provided by the run-time support package nested within the KAPSE.

Communication takes place via entries. If task A wants to communicate with task B, then task A must call an entry of task B. This communication is assymetric. That is, task A must know that it is calling task B but task B does not necessarily know that it is being called by task A. Note that this is a departure from the CSP model. In fact, each entry of each task has associated with it a queue and callers of the entry are placed into the queue in a first in first out order. When task A calls task B and task B acknowledges the request (via an Ada "accept" statement) then A and B are said to be in rendezvous.

In the simplest form of rendezvous, whichever task (caller or server) gets to the rendezvous point first goes into a sleeping wait. When the other gets to the rendezvous point, the first wakes up and rendezvous occurs. While this rendezvous is taking place, the two tasks are locked together (there is a single thread of execution). Once rendezvous is completed, the tasks are free to continue concurrently (there are multiple threads of execution).

Some of the features available with the Ada tasking model:

A server task can select one of several of its entry queues to serve.

A server task can either go into a sleeping wait or perform some indivisible sequence of statements if all of its entry queues are empty.

A server task can serve any one of its entry queues if that queue becomes nonempty within a specified time. If not, it can perform some alternative action.

A server task can choose to consider certain entry queues only if some particular condition (guard) is true.

A server task can offer to terminate if all of its queues are empty.

A calling task can choose to wait only a specified time for service.

A calling task can choose to rendezvous if and only if it can do so immediately.

A task can be spawned off dynamically.

A task can suspend itself for some finite period of time.

Thus, it can be seen that the Ada tasking model contains a very rich set of primitive operations for concurrency. From this collection, monitors, semaphones, schedulers, etc. can easily be implemented.

GENERIC DEFINITION

Perhaps the best analogy for describing Ada's generic definition capability is that a generic unit in Ada is very similar to a macro in assembly language. The Ada generic capability provides parameterization of subprograms and packages (there is no such thing as a generic task) with types, objects and subprograms.

If we consider a subprogram, we see that it really represents a factorization of similar algorithms. That which is consistent (the subprogram body) is factored out from that which differs (the subprogram parameters). The resolution of actual/formal parameters occurs during run time. Analogous to this is the factorization that occurs with generic units. That which is consistent (the generic body) is factored out from that which differs (the generic parameters). However, the resolution of actual/formal parameters occurs prior to actual execution. Further, only values and objects can be passed as actual parameters to a subprogram. But, types and subprograms, in addition to values and objects, can be passed as actual parameters to generic units. It is as if these actual parameters are used to construct a unit which uses the generic unit as a template. This construction is known as generic instantiation.

If we consider the abstract object known as a stack and the abstract operations of push and pop, we have a very powerful data structure. Suppose we want a stack of integers and a stack of automobiles. Clearly these two situations differ only in the data type of the components of the stack. As in most languages, it is not possible to pass a data type to a subprogram at run time. This is clearly an opportunity for using generic units. The template for the generic package could be written in terms of some generic parameter (say ELEMENT_TYPE). Then two specific instantiations could be given. One where type INTEGER is systematically substituted for ELEMENT_TYPE and another where type AUTOMOBILE is substituted for ELEMENT_TYPE at each occurence. We would now have two stack packages just as if we had written them from the very beginning.

It is envisioned that Ada's generic definition capability will be the cornerstone of an industry of off-the-shelf, reuseable software components.

EXCEPTION HANDLING

Ada provides the application programmer the opportunity to "trap" exceptional conditions (like division by zero, exceeding array bounds, etc.) rather than have the operating system deal with them as is generally the case. This provides yet another example of shielding the software developer from the operating system.

Exceptions can be raised implicitly by the system when errors occur or explicitly by the programmer when an error condition is sensed. Further, user-defined exceptions (such as STACK_OVERFLOW or CHANGE_BARREL_IS_EMPTY) can be dealt with as well as system-defined exceptions (such as NUMERIC_ERROR, CONSTRAINT_ERROR, etc.) Of course, user-defined errors can only be raised explicitly.

Exceptions are handled relative to a frame of reference. A frame of reference can be a block statement, subprogram body, task body or package body. Each of these can contain a sequence of statements and, at the very end of the sequence of statements, an exception handler. This handler contains statements which will be executed only if an exception is raised during execution of the sequence of statements of the frame of reference. If there is no such handler, then the exception is propagated. For example, if an exception is raised inside of a block statement then execution is abandoned and control is transfered to the exception handler. If there is no exception handler then the exception is propagated by raising it again at the next sequential statement following the block statement. If the same situation arose inside of an active subprogram body, then the exception would be propagated by raising the same exception at the point of call of the subprogram.

The Ada exception handling capability allows a very nice separation of steady_state logic from error checking logic. In addition, the ability to raise and handle a DATA_ERROR when erroneous data is input, allows the fairly straightforward input (and output) of enumerated values. This is one of several shortcomings in the Pascal language which are eliminated in Ada.

REPRESENTATION SPECIFICATION

Transportable software has been one of the goals of the Ada movement since its inception. However there are often aspects of software, especially mission critical software, which run counter to transportability. Often, because of external interfaces to peripheral hardware, or to provide more efficient representation of internal objects, it becomes necessary to make certain decisions to take advantage of certain capabilities of the host or target architecture. This "bit-twiddling" is often decried as heinous but is sometimes necessary.

Ada allows certain machine-dependent activity. For example, the applications programmer can dictate the underlying representation for enumeration type literals, the actual size reserved for objects of a given type, the actual memory location where a given object or subprogram begins, the amount of storage reserved for dynamic allocation, the actual, bit-by-bit description of objects of a record type (excellent for dealing with Program Status Words and the like), etc. Also, if a given architecture will vector to a certain address upon receipt of a hardware interrupt, that actual address can be the location of a task entry point and can be indicated by a representation specification.

It is important to realize that representation specifications are not currently part of the validation suite. Therefore, an implementation is not required to include these specifications.

INTERFACE TO OTHER LANGUAGES

"A subprogram written in another language can be called from an Ada program provided that all communication is achieved via parameters and function results."[16]

There is a great amount of software which has been written, tested and, in some cases, proven to be correct. Throughout the Ada movement it has never been the intent that existing software be rewritten in Ada. Therefore the language makes provision for calling subprograms which have actually been written in other languages.

This capability is not required by an implementation (it is not part of the validation suite testing). Current experience shows that the most common language supported by this mechanism are FORTRAN and the assembly language associated with the underlying architecture.

EUROPEAN PARTICIPATION WITH ADA

Although Ada was initiated by the U. S. Department of Defense, there seems to have been as much Ada activity in Europe as in the United States. The language was developed by a primarily European team and interest in the language continues at a high level. The following is a sample of European Ada involvement: All information has been gleaned from the Ada Information Clearing House.[17]

During the summer of 1985, NATO issued a mandate requiring the use of Ada for all NATO-developed systems. This mandate took effect 1 January 1986.

Also during summer, 1985, an agreement was reached with the Ministry of Defense to establish a third validation facility (France and Germany having the first two) in Europe. The difference is that the UK will be able to issue its own validation certificates rather than have the certificates issued by the US.

In March, 1985 it was announced that the Commission of European Communities (CEC) had contracted with Dansk Datamatik Center (DDC) and Consorzio per la Ricerca de le Applicazioni di Informatica (CRAI) to develop a draft formal definition of the standard American National Standards Institute (ANSI) Ada. In addition to the Danish and Italian principle organizations, the CEC will establish a review group from approximately 12 countries in Europe, the U. S. and Canada.

In January, 1985 Informatique Internationale, of France delivered a queuing network simulation program to the Spanish company CESELSA. The program was written entirely in Ada and is appropriate for any queuing system including inventory management, ground and air traffic monitoring and production planning and manufacturing.

In December, 1984, ALSYS, a French company headed by Jean Ichbiah (the principle developer of the Ada language) was awarded a validation certificate for its AlsyCOMP 0001 version 1.0 Ada Compiler. Host machines included the Vax 11/785, 11/780, 11/750, 11/730 and the Micro Vax 1 with host operating system is the VMS version 4.0. The target system is the ALTOS ACS 68000 under the ALTOS Operating System version 1.

In July, 1984 the UK announced that the Ministry of Defense would require the use of Ada in real-time computer applications. They stated "Confidence in the Ada language is now such that we can announce our intention to adopt it as a successor to CORAL for real-time defense applications." In general, projects initiated prior to 1 July 1987 can choose Ada or CORAL but projects initiated after that date will use Ada.

In November, 1984, The University of Karlruhe, under contract to the Ministry of Defense of the Federal Republic of Germany was awarded validation certificates for two Ada compilers. The first Ada compiler is the Siemens BS2000 version 840404. Its host and target is the Siemens 7.571 under the BS2000 7.1 operating system. The second compiler is the Vax 11 Version V1.0. The host and target is the VAX 11/750 under VMS version 3.0.

In October, 1984 Dansk Datamatik Center (DDC) was awarded a validation certificate for its DDC Ada Compiler VAX 11 Release 1.1. The DDC Ada Compiler host and target machine is the VAX 11/750 under the VMS 3.5 operating system.

York University is writing an Ada compiler for Vax systems under UNIX.

In addition to the above activity, many of the finest textbooks on the Ada language have come from European authors.

ADA EXPERIENCE

The following represents findings from various organizations currently engaged in Ada activity. No attempt has been made to validate the findings and no attempt has been made to find a common level of comparison among the findings. Except where noted, all information is derived from an Ada Joint Program Office briefing.[18]

Bell Technical Operations of Tucson, Arizona developed a commercial application with over 103,000 lines of Ada code. They feel that the judicious use of the Ada generic facility reduced the total lines of code by approximately 40%.[19]

MacDonnell-Douglass Aircraft Company of St. Louis, Missouri, developed a Computer-Based Training system which included almost 500,000 lines of Ada code. They found that it was almost trivial to rehost this system from a Vax-11/780 to a Gould SEL and finally to a Pacific Micro 68020 based system.[20]

MacDonnell-Douglass also found, in their F-15 flight control experience, that they achieved a five-to-one productivity improvement over that realized with assembly language. Further, they experienced only a 10% cost in execution time and a 36% cost in memory.

Northrop Aircraft Corporation of Hawthorne, California noticed a five-to-four reduction in source lines from Jovial J73 to Ada.

IBM's Federal Systems Division, using Ada as PDL on a major system for the Sattelite Control Facility reports the following findings:

--30% improvement in programmer productivity

--15-20% problem reports/1000 lines
--95% success rate in initial problem fixes
--Only 1% of fixes propagated new problems

Digital Electronic Systems of Esthill Springs, Tennessee, working on various configuration control systems, reports productivity of 2,241 lines per man month and estimate that 95% of their software was successful after the initial compilation.

Intellimac of Rockville, Maryland indicates 800-1200 lines per man month. The 1200 figure is reported also by Moog Corporation of Buffalo, New York.

Raytheon Corporation of Mt. Laurel, New Jersey found that in transporting a system from a 6800 micro to a Vax, only 6 line changes per 12,500 Ada lines were required.

It is clear that there is wide-spread support emerging for the Ada language. It is now time for serious research into the productivity claims in order to verify the numbers and to make estimates which range across experience base and application areas.

CONCLUSION

The Ada programming language is the result of a carefully considered design approach which was, at every juncture, submitted for public review by users, implementors and academicians alike. It is quite likely that this language, with its support environment, will be a very helpful instrument for achieving transportability of software, tools and even people in the years to come. Also, initial data seems to indicate that software developers can be more productive using the Ada language than they were using previous languages. Finally, the use of Ada will in no way guarantee the generation of better software. The undisciplined use of this language will likely result in software which is just as difficult to maintain as our existing efforts. However, if good, modern principles of software engineering are used in designing our sofware, the Ada language will enforce those decisions better than any other language to date. Perhaps most importantly, almost everyone who has seriously tried the language, likes it.

BIBLIOGRAPHY

1. Dijkstra, E.W., "The Humble Programmer" (Turing Award Lecture, Communications of the ACM, Vol. 15, No. 10 (October 1972): 861.

2. Fisher, David A., "DoD's Common Programming Language Effort" IEEE Computer (March 1978): 24

3. Mission Critical Computer Resources Definition, Title 10, U.S. Code 2315 (the Warner Amendment), Ada Information Clearinghouse Newsletter (November 1984).

4. Fisher, David A., p. 25

5. Stephen, D.G. et al, "DoD Digital Data Processing Study; A Ten Year Forecast," Electronics Industries Association, 1980.

6. Whitaker, William A., "The U.S. Department of Defense Common High Order Language Effort," AC, Sigplan Notices, (February 1978): 22

7. Fisher, David A., p. 26

8. Requirements for High Order Programming Languages, STEELMAN, Department of Defense, June 1978.

9. Booch, Grady, Software Engineering with Ada, Benjamin /Cummings, Menlo Park, California, 1983.

10. Stenning, Vic et al, "The Ada Environment; A Perspective," lEEE Computer, (June, 1981): 26.

11. Locke, Doug, "The Ada Programming Support Environment," IBM Software Engineering Exchange, (October 1980):21.

12. Ibid.

13. Buxton, John N. and Druffel, Larry E. "Requirements for an Ada Programming Support Environment: Rationale for STONEMAN," Proceedings, Compsac 80, (October 1980): 68

14. Ross, D.T. et al, "Software Engineering: Process, Principles, and Goals," Computer (May 1975): 65.

15. Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, (August 1978): 66

16. Reference Manual for the Ada Programming Language, United States Department of Defense, Ada Joint Program Office, July 1982.

17. Ada Information Clearinghouse, 3D139 (1211FERN,C-107), The Pentagon, Washington, DC 20301-3081.

18. Kopp, Alan, "1980's Ada Goldrush", briefing presented at Ada/UK Conference, University of York, January 1985.

19. Arndt, Douglas, "The Application of Ada Generics to Large-scale Projects: A Case Study:, internal paper, Bell Technical Operations Corporation, Tucson, Arizona.

20. Bolz, Richard and Pflaster, Dave "A Computer Based Training System Written in Ada", briefing presented at Ada-Jovial User's Group meeting, 26 February 1985, San Jose, California

"PARALLEL PROCESSING" AND ADA * TASKS

T. P. Baker

Department of Computer Science
The Florida State University
Tallahassee, FL 32306
U.S.A.

December 16, 1985

SUMMARY

This paper explores some of the possible uses and limitations of Ada tasks within the context of real-time embedded computer applications. It also surveys the extent to which tasks and some other Ada features map naturally onto physically parallel computers. Some familiarity with Ada, and in particular Ada tasks, is presumed, though the paper does review some aspects of Ada task semantics.

1 INTRODUCTION

One of the more distinctive features of the programming language Ada is its provision for defining multiple tasks. The Ada Standard says: "Tasks are entities whose executions proceed in parallel in the following sense. Each task can be considered to be executed by a logical processor of its own. Different tasks (different logical processors) proceed independently, except at points where they synchronize ...Parallel tasks (parallel logical processors) may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can detect that the same effect can be guaranteed if parts of the actions of given task are executed by different physical processors acting in parallel, it may choose to execute them in this way; in such a case, several physical processors implement a single logical processor." [1]

That is, tasks are logically parallel divisions of a program. Whether there is any physical parallelism in the execution of an Ada program is entirely independent of its division into tasks.

In real-time embedded systems, both kinds of parallelism are of interest. Physical parallelism is of interest because it can provide greater processing speed than sequential processing. Logical parallelism is of interest because it is a natural characteristic of real-time applications.

2 PARALLELISM IN A REAL-TIME SYSTEM

To illustrate logical parallelism and the motivation for physical parallelism in a real-time application, we consider an imaginary simple aircraft tracking system, shown in Figure 1. The inputs to this system are from a tracking (e.g. phased array) radar and an operator console; the outputs are to the radar and to a graphic display device viewed by the operator. In addition to using radar data to keep the information on the display up to date, the system maintains an internal table or "track file", showing the estimated current position of each tracked object, for use by the tracking system itself and by other systems which it supports.

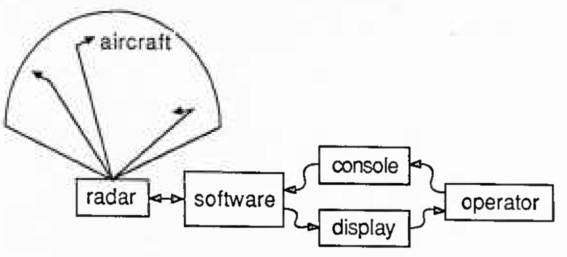


Figure 1: A simplified aircraft tracking system.

The tracking system software must meet the following requirements:

- Each command going to the radar specifies a cell of space in which the radar is to look. The radar buffers a limited number of commands internally, and executes them in order of arrival. Determining whether the radar is ready for another command requires polling the radar command port at least once every R seconds.
- Reports from the radar arrive over a direct memory access (DMA) channel. The radar generates an interrupt whenever it fills its current report buffer. The tracking system must respond by giving the radar the location of a free buffer. This must be done promptly, or data will be lost. Each report describes a set of objects sighted by the radar in a single cell. If a command results in no object being sighted, there is no report. If more than one object is sighted in a particular cell, the radar returns a report describing all the objects.

*Ada is a registered trademark of the U.S. Department of Defense (AJPO).

- The positions of all tracked objects are to be updated in the track file at least one every T seconds, using fresh data from the radar, if possible.
- The radar is to be used both to keep known objects in track, and to search for new objects, giving priority to keeping known objects in track.
- The graphic display is a DMA device, which periodically refreshes its screen from an image stored shared-access memory. All the information in this area is to be updated at least once every D seconds, using current data from the track file.
- The operator console generates an interrupt whenever the operator hits a key. Such interrupts are to be handled immediately. As soon as a complete console command has been accumulated it is to be executed within C seconds.
- The operator can add annotations to tracked objects and direct the searching to concentrate on areas of particular interest.

The work which the application software must do to meet these requirements is most naturally viewed as a set of logically parallel activities, which must go on more or less asynchronously and at independent rates. (See Figure 2.) In Ada, these activities can be described as a collection of tasks, within a single program. We will discuss how such a system might be programmed in Ada, further on in this paper.

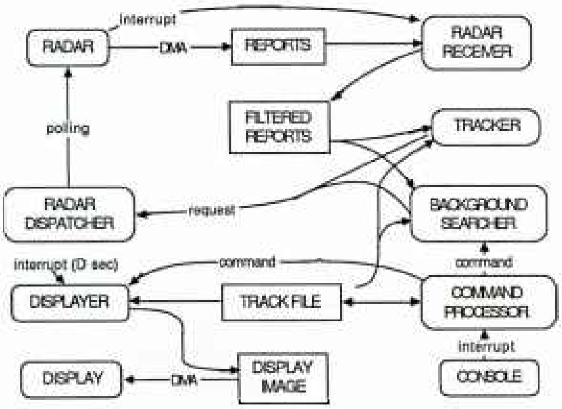


Figure 2: System activities

This example also presents a situation where a software designer may need to deal with physically parallel processing. In order to keep up with the radar, which we assume is capable of tracking many objects simultaneously, and to discriminate objects from noise, a great deal of computation will need to be done quickly. Suppose no single processor capable of doing all these computations quickly enough is available. The hardware designers therefore propose a configuration with several processors and several shared memory banks, such as that shown in Figure 3.

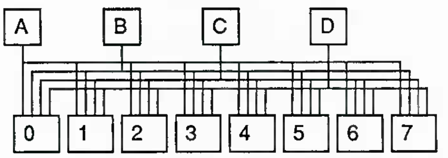


Figure 3: A multiprocessor configuration.

Although in 1985 no multiprocessor Ada compilers and runtime systems have yet been developed, it appears that such a collection of processors can be used to achieve physically parallel execution of Ada tasks. We will consider how this might be done, after a quick review of Ada task semantics.

3 TASKS IN ADA

Ada tasks are data objects. They follow the normal scope and lifetime rules of data objects. Like all data objects, each task has a type. All tasks of a type share the same executable code. Each task has its own storage to hold the values of data within it. Unlike other kinds of data objects, tasks are also executable program units. A task defines a level of static nesting. It may have local declarations, and code within it may access both local and nonlocal declarations. Thus several tasks may share read and update access rights to variables, and allocation rights to a collection of objects designated by an access type. Tasks may be nested within recursive procedures and within other tasks. They also may have local exception handlers.

When a task is created it is inactive. Later, it begins a period of execution, called "activation", during which it initializes local data. After its activation is complete, and possibly after some waiting to synchronize with other tasks, the task begins normal execution. During normal execution it may communicate with other tasks, by means of "rendezvous" and shared variables. From time to time, execution of the task may be forced to wait for some event. Execution may continue indefinitely, or it may eventually complete. After completion, the task and its storage may still be referenced by other tasks. Finally, the task may "terminate", at which point its storage is no longer needed, and for all nontrivial purposes the task may be considered to no longer exist. Figure 4, below, shows these principle state transitions of a task.

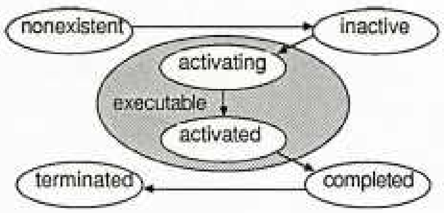


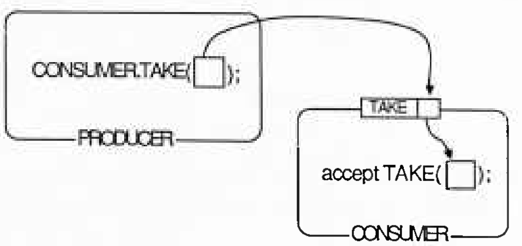
Figure 4: Important task state distinctions.

Tasks are essentially asynchronous and nondeterministic. They may be created at arbitrary times, and go through their lives independently, except at a few points:
(1) two tasks participating in a "rendezvous" are synchronized at the start and end of the rendezvous;
(2) a task is synchronized with the task that created it at the start and end of its activation;
(3) a task that has completed its execution is synchronized with every other task.

A rendezvous takes place when one task calls an entry of another task, and the other task expresses readiness to accept a call on that entry, via an accept or selective wait statement. From the calling task's point of view, an entry call is syntactically and semantically like a procedure call (even with respect to details such as the propagation of exceptions raised and not handled during the call). As with a procedure call, the execution at the point of call is suspended until the call is complete. During the rendezvous, parameters may be passed between the two tasks. From the acceptor's point of view, however, the rendezvous is different. The acceptor cannot in general determine the identity of the calling task, and must accept calls on each entry in "first-in-first-out" (FIFO) order.

Note that in order for a rendezvous to take place, the calling task must be able to name the task it calls. This means that the called task must be visible at the point of call, or must be named indirectly, via an access value. Because it is referenced in the call by name, the type of the called task is fixed at the time the entry call is compiled. In contrast, from the point of view of the accepting task there is no restriction on tasks that may call one of its entries, other than that they be able to name the entry. This asymmetry must be taken into account when designing with tasks. For example, it means that a "server" task in general cannot "call back" those tasks which it serves, since it has no way of naming them.

As an example of an Ada program with tasks, and rendezvous in particular, consider the following pair of producer-consumer tasks. PRODUCER calls CONSUMER's entry TAKE whenever it has a datum to transmit, and CONSUMER accepts a call on TAKE whenever it is ready to receive a datum. Whichever one attempts to make the rendezvous first will wait until the other is ready.




```
procedure MAIN is . . .
        task PRODUCER;                                          -- Task PRODUCER created.
        task CONSUMER is entry TAKE(X: in out T); end CONSUMER; -- Task CONSUMER created.
        task body PRODUCER is X: T;                             -- Producer finishes activation.
        begin loop . . . CONSUMER.TAKE(X);                      -- Calls entry TAKE of CONSUMER.
                 . . .
            end loop;
        end PRODUCER;
        task body CONSUMER is . . .                             -- CONSUMER finishes activation.
        begin loop accept TAKE(X: in out T) do;                 -- CONSUMER accepts entry call.
                      . . .
                   end TAKE;                                    -- Rendezvous is completed.
                 . . .
            end loop;
        end CONSUMER;                                           -- Procedure body MAIN finishes elaboration.
                                                                -- PRODUCER and CONSUMER start activation.
begin . . .

end MAIN;                                                       -- Procedure MAIN completes execution.
```

Figure 5: Producer and consumer tasks.

Figure 6 illustrates how these three tasks might execute in parallel, given three separate processors.

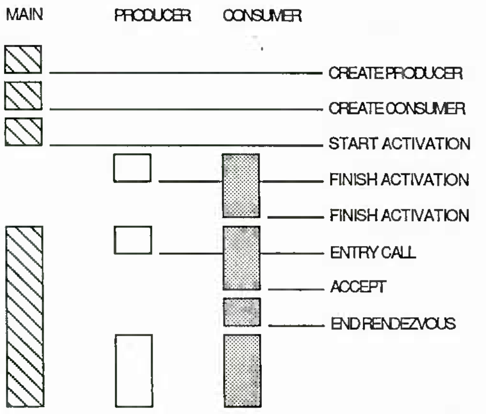


Figure 6: Tasks executing in parallel.

Operations that require synchronization of tasks, or mutual exclusion between tasks, such as allocation out of a shared pool of storage, are likely to be implemented by a collection of software provided with an Ada compiler, that must be coresident with compiled programs at run time. We call this collection of support software a "runtime environment". Other support services provided by the Ada runtime environment include maintenance of a real-time clock, task dispatching, and any aspects of storage management and exception handling that are not implemented directly by compiler-generated code. The runtime system may also be responsible for linking interrupts to handlers within the Ada program.

The runtime environment is most efficiently hosted on a "bare" machine – that is, one without any other operating system. In this case all functions typically provided by an operating system or executive are provided either by the Ada runtime environment or by application code (written in Ada). Because it need not support many of the more complex functions of a traditional operating system, such as providing a file system, an Ada runtime environment can be quite small. For example, in the FSU/AFATL implementation it occupies less than 8K bytes of memory.

A real-time embedded systems builder is likely to be interested in details of an Ada implementation, and in particular the runtime environment, because they determine execution timing. Unfortunately, the details of interest will vary from one compiler to another. Even the division of duties between inline code (generated by the compiler) and the runtime environment may vary for one compiler, according to the level of optimization. Thus real-time programmers using Ada may need to learn to survive with less control than they are accustomed to exercising over things like execution timing and storage utilization.

4 MAPPING ADA ONTO PARALLEL MACHINES

There is considerable interest in using Ada to develop software for parallel hardware configurations. Parallelism in computer hardware comes in a number of different forms. Among the various forms of parallel computer architectures are:

- Pipelined scalar processors, which partially overlap the execution of successive instructions within a sequence.
- "Vector" or "array" processors, which can rapidly perform a single operation on a sequence of data, using pipelined or fully parallel execution.
- Multiprocessor configurations with shared memory, where each processor executes its own independent instruction sequence in parallel with the rest, but communication and synchronization are possible through a shared (arbitrated) memory.
- Multiprocessor configurations without shared memory but linked by a communications network.

Among these, the finer granularity of parallelism offered by pipelined and array processors distinguishes them sharply from multiprocessor configurations. Ada does not provide any standard means of expressing such fine-grained parallelism explicitly. Nevertheless, an optimizing compiler may be able to find opportunities for such parallelism within a program. That is, the code generator of a comnpiler targeted to a pipelined processor might be designed to take advantage of the pipelining. A compiler might also do loop-conversion and other vectorizing transformations similar to those currently done by "vectorizing" FORTRAN compilers. It is likely to be some time, however, before we see Ada compilers reach this level of sophistication, since in general the complexity of the Ada semantics make optimizations difficult. For the case of vector processors, there is an alternative to compiler-supplied optimizations. Explicit access to hardware parallelism can be provided through an implementation-defined package of vector operations.

In contrast, coarser grained parallelism is explicitly representable in stanadard Ada – through tasks and through multiple programs. Physically parallel execution of tasks is likely to map more efficiently onto some hardware architectures than others, however. Because tasks must be able to share read and write access to common data, and because intertask communication requires rendezvous, the Ada task model seems best suited to architectures with high-bandwidth interprocessor communication and low-cost synchronization.

For instance, it appears quite practical for several processors that share access to a common memory (such as those shown in Figure 3) to execute different tasks within a single program, in parallel. It also appears practical to distribute Ada tasks over a system of more loosely connected processors. The notion of rendezvous (i.e. remote procedure call) seems fairly well suited to the latter environment, and has been proven workable in network operating systems. Shared access to non-local data is still possible, using remote read and write operations. The main problem seems to be efficiency. Still, in 1985, no such multiprocessor implementations of Ada have yet appeared.

Unfortunately, for some architectures, such as closely coupled multiprocessor systems with very many processing elements, neither the task nor the program seems to be a suitable unit of granularity. It may be argued that Ada was not designed to support such architectures. If Ada programs are to be mapped onto such machines in such a way as to take full advantage of their power further research advances appear necessary.

One direction such research is taking, exemplified by work done at Honeywell [2], is that parallelism introduced solely for speed or redundancy should not be expressed directly in Ada. In this view the logical design of a program should be expressed in its simplest and most natural form, without regard to the hardware architecture on which the program will be executed. Details of how a program is mapped onto particular hardware should be expressed in a separate specification. While this approach appears very desirable from the point of view of maintaining independence of hardware and software, its feasibility remains to be demonstrated.

Given today's compilers it seems most practical to view each processor as executing a separate program (or several). Such programs can communicate by means of common interfaces, implemented as shared library-packages. In addition to being immediately practical, this multiprogram approach has another advantage over the single-program model – it supports an "open" system architecture. Unlike communicating tasks, which must be compiled together within a single program, separate programs that communicate via common packages may be compiled and loaded independently. Moreover, communication linkages may be changed dynamically, during execution.

5 SOME BASIC DESIGN CONSIDERATIONS

Before returning to the radar tracking example, we examine some of the basic problems in designing real-time software with Ada tasks, and some useful techniques.

5.1 SHARING A PROCESSOR

When there is more than one task ready to be executed by a single physical processor the Ada language implementation must choose one of the tasks to be executed. We call this process of allocating processor time to tasks "dispatching". The Ada Standard leaves the details of dispatching to be determined by an implementation. This is good, because it gives the implementation freedom to make efficient use of the particular hardware resources that may be available – e.g., multiple processors. On the other hand, it is a problem for real-time system designers, who need to be able to predict system performance.

A programmer can exert limited control over dispatching by means of static priorities, which may be assigned to task types. Whenever several tasks are competing to be executed by the same processor, the processor must be allocated to one with highest priority. Additional control may be provided by associating entries of tasks with hardware interrupts, since while a task is handling an interrupt it has effectively higher priority than any task that is not handling an interrupt.

An unpleasant consequence of Ada's leaving the detials of dispatching unspecified is that programs with multiple tasks are likely to depend on implementation features for their correctness. Consider, for example, the two tasks shown below.

```
task URGENT is pragma PRIORITY(100); end URGENT;
task LESS_URGENT is pragma PRIORITY(0); end LESS_URGENT;
task body URGENT is
begin loop . . .                                  -- Do urgent work.
          delay 0.1;
     end loop;
end URGENT;
task body LESS_URGENT is
begin loop . . .                                  -- Do less urgent work.
     end loop;
end LESS_URGENT;
```

Figure 7: Controlling dispatching with priorities and the delay statement.

Let us look at this example from a pessimistic point of view, assuming the Ada implementation is validated, but no more. Several things may go wrong:
(1) The compiler may ignore the pragma PRIORITY.
(2) Among tasks of equal priority, the runtime environment may give dispatching preference to the currently active task. That is the dispatching policy may be "unfair".
(3) The runtime environment may not notice that task URGENT's delay has expired, until it is invoked for some other reason. That is, the delay statement may be implemented in a non-preemptive manner.
If (2) holds, and (1) or (3) also holds, task LESS_URGENT will "starve out" task URGENT. All of the above are permitted by the present validation tests (ACVC Version 1.6).

It appears that (3) will be disallowed in the future. That is, while the Standard only requires that a task that executes a delay statement be suspended for "at least" the duration specified, there must be some limit to how long it is suspended, and when the delayed task is released it should preempt a processor from any lower priority task. Still, the accuracy of the the delay is unpredictable, and in an implementation seeking to minimize worst-case overhead it may become less accurate as the number of tasks increases. Finally, unless the delayed task has higher priority than every other task, it may wait an unpredictably long time before its execution resumes.

Some would argue further that a good implementation would not allow (2), but we do not agree. Switching a processor from one task to another unnecessarily is a waste of the processor's time.

Designing an Ada program involving tasks so that it is free of implementation-dependencies related to dispatching is not only difficult, it is probably also unwise. A program that is totally implementation-independent is likely not to permit efficient use of multiprocessor hardware configurations. In particular, to make certain that a system of tasks can be executed on a single processor with an unfair dispatching policy it appears necessary to structure the program in such a way that for each task there is some time when all other tasks of higher or equal priority voluntarily wait, so that that task can be guaranteed to execute. For configurations with more than one processor this would mean that all but one of the processors would be idle much of the time.

5.2 PROBLEMS WITH RENDEZVOUS

The normal means of intertask synchronization and communication in Ada is by means of rendezvous. This presents two problems for a designer of real-time software systems: overhead and waiting.

5.2.1 OVERHEAD

The first problem is that rendezvous is a fairly complex, and therefore fairly costly operation. Because an entry call is similar in efect to a procedure call, we ran a simple test to compare the execution times of these two constructs, using our own (FSU/AFATL) Ada compiler – a single-processor implementation for the Z8002 microprocessor – and the test program in Figure 8. It has two tasks, one of which loops, repeatedly making a rendezvous with an entry of the other. In one second, 366 iterations were completed. In a comparison test, where the rendezvous was replaced by a procedure call, 957 iterations were completed. Replacing the procedure call by inline code resulted in a further increase, to 1081 iterations. Solving the three linear equations representing the running time of the loop, we find that it takes 15 times longer to perform a rendezvous service cycle (with no parameters) than to perform a similar procedure call and return. Even though the speed of rendezvous in the FSU implementation could be improved, so could the speed of procedure calls. Therefore this is probably a fair estimate of the overhead of rendezvous.

In such a single-processor implementation, the additional cost of the rendezvous over the procedure call is largely due to making the one task wait for the other, determining when the dynamic conditions necessary for rendezvous have been met, and switching the context of execution between tasks. In a multiprocessor implementation (when such become available) there would be additional overhead, due to interprocessor communication. Though claims have been made [3] that compiler optimizations can dramatically reduce the average cost of rendezvous by recognizing special cases ("idioms"), the feasibility and practical benefits of such optimizations have yet to be demonstrated.

```
procedure TIMER is . . .
        I: INTEGER:= 0;
        task A is entry GO; end A;
        task B is entry GO; entry E; end B;
        procedure REPORT is begin put_line(I); abort A,B; end REPORT;
        task body A is
        begin accept GO;
              loop B.E;                              - - This entry call is replaced by a procedure call or inline code.
                    NOW:=CLOCK; D:=NOW-START;
                    if D >= 1.0 then REPORT; end if;
              end loop;
        end A;
        task body B is
        begin accept GO;
              loop accept E do I:=I+1; end E; end loop;
        end B;
begin START:=CLOCK; A.GO; B.GO; end TIMER;
```

Figure 8: Program used to measure relative execution time of rendezvous.

The overhead of rendezvous should be considered in the design of a software system. In particular, task interactions should be designed so that the work done between rendezvous should counterbalance the cost of the rendezvous. For example, if a rendezvous takes 15 time longer than a procedure call, and the work done between rendezvous is at least equivalent to 300 procedure calls, the overhead of tasking due to rendezvous will be limited to 5 percent.

5.2.2 WAITING

The second problem with rendezvous is that one of the tasks always must be willing to wait. The limitations that this need for waiting imposes on task communication can be seen by reconsidering the producer-consumer example, discussed above. So long as either PRODUCER or CONSUMER can always afford to wait there is no problem, but in real-time applications this is not always the case. Suppose, for example, that PRODUCER is an interrupt-driven task, and CONSUMER is a periodic task, that must execute exactly once every T seconds, using any new information that PRODUCER has passed to it. Alternatively, suppose that PRODUCER is executing on one physical processor and CONSUMER is executing on another; if either task must wait, valuable processor time may be wasted.

To avoid or limit waiting, a task that wishes to make a rendezvous might use a conditional or timed entry call, or a selective wait with an else or delay clause. If the rendezvous is not begun within the specified delay the task is released and may go on with its execution. The task does not have to wait, but then neither does it accomplish the purpose of the rendezvous. For example, suppose that the task PRODUCER used a conditional entry call to pass data to the task CONSUMER, as shown in Figure 9. Because PRODUCER may need to produce more than one datum between successful rendezvous with CONSUMER, the two tasks now communicate through a buffer that can hold more than one datum.

```
task body PRODUCER is
    BUFFER: array ( . . . ) of T;
begin loop
            . . .                                     - - Do some work; put datum into buffer if it fits.
            select CONSUMER.TAKE(BUFFER);
            else null;
            end select;
      end loop;
end PRODUCER;
```

Figure 9: Using a conditional entry call to avoid waiting.

There are several problems with this proposed solution:

- CONSUMER must be willing to wait; otherwise there may never be any rendezvous. (This is the main problem.)
- Executing the conditional entry call takes time, which is essentially wasted in the case that no rendezvous is made.
- If the buffer ever fills up entirely, PRODUCER can do no more useful work. (This is unavoidable, if PRODUCER does not wait, but we hope the buffer is big enough.)
- Since an Ada implementation is not required to do time slicing, if PRODUCER shares a processor with CONSUMER it may starve out CONSUMER. (This is really an independent problem, and does not arise at all if PRODUCER has a separate processor.)
- CONSUMER must copy all the data from the buffer to a working area, so that it can release PRODUCER from the rendezous quickly. (This is also an independent problem, which is easily overcome using indirect buffers, as shown further below.)

```
task body PRODUCER is
    BUFFER: array ( . . . ) of T;
begin loop . . .                                     - - Do some work; put datum into the buffer, if it fits.
            select CONSUMER.TAKE(BUFFER); else delay 0.1; end select;
      end loop;
end PRODUCER;
```

Figure 10: Using a timed entry call to limit waiting.

A timed entry call could also be used, as in Figure 10 . Here PRODUCER delays itself for 0.1 second if it cannot make the rendezvous. The chief advantage of this is that it may let other tasks on the same processor do some work. However, we have not gained much:

- CONSUMER still must be willing to wait.
- PRODUCER now also may need to wait (though if it has to wait very long it will wake up and do some other work). This solution would therefore be inappropriate if PRODUCER were interrupt-driven.
- Still more execution time (due to scheduling the wakeup event for PRODUCER) will be spent in the case that the rendezvous cannot be made immediately.

For each of the solutions above there is also a complementary solution, in which the accepting task limits its waiting time, using a selective wait statement with a delay alternative or else part. Though the roles of the two tasks are reversed, the limitations are the same.

5.3 USING SHARED VARIABLES TO AVOID RENDEZVOUS

It might seem that we would be better off if we could avoid rendezvous entirely. Sometimes this can be done safely by means of conventional buffering techniques, making use of shared variables and the pragma SHARED. Using the pragma SHARED we can specify that every read and update of a particular simple variable of a scalar or access type is effectively an atomic operation. Consider the example in Figures 11 – 13 , in which the producer and consumer communicate via indirect buffers, using shared access variables to communicate their current positions in a chain of shared buffers.

```
generic type DATA_TYPE is private;
        SIZE: in INTEGER;
package QUEUE is
        - - Assumes each instantiation of this package is "owned" by a single writer task,
        - - so that the list of available buffers need not be synchronized.
        - - Each instantiation may be accessed by one reader task.
        - - This reader task should only call ADVANCE and read MARK.all.
        type BUFFER;
        type BUFFER_ACCESS is access BUFFER;
        type BUFFER is record NEXT: BUFFER_ACCESS; DATA: DATA_TYPE; end record;
        HEAD, MARK, TAIL: BUFFER_ACCESS:= null;
        - - HEAD is accessed only by writer; it is used to collect used buffers.
        - - MARK is the buffer currrently begin read. It is read by writer and updated by reader.
                pragma SHARED(MARK);
        - - TAIL is the buffer currently being written-to. It is read by reader and updated by writer.
                pragma SHARED(TAIL);
        procedure READER_ADVANCE(OK: out BOOLEAN);
                - - Move MARK to next full buffer in chain, if any. Set OK=true iff MARK is advanced.
        procedure WRITER_ADVANCE;
                - - Add new (empty) buffer to tail of chain, pushing current tail
                - - buffer into range of full buffers, that reader may access.
end QUEUE;
```

Figure 11: The specification of a generic queue package, using shared variables.

```
package body QUEUE is
        AVAIL: BUFFER_ACCESS:= null;
        procedure READER_ADVANCE(OK: BOOLEAN) is
        begin if MARK = TAIL then OK:= false; else OK:=true; MARK:= MARK.NEXT; end if;
        end READER_ADVANCE;
        procedure WRITER_ADVANCE is
                T1: BUFFER_ACCESS:= null;
                T2: BUFFER_ACCESS:= HEAD;
        begin while T2 /= MARK; loop T1:= T2; T2:=T2.NEXT; end loop;
              if T1 = null
              then if AVAIL = null then T1:= new BUFFER else T1:= AVAIL; AVAIL:= AVAIL.NEXT; end if;
              elsif T1 /= HEAD then T1.NEXT:= AVAIL; AVAIL:= HEAD.NEXT; end if;
              HEAD:= T2; T1.NEXT:= null; TAIL.NEXT:= T1;
        end WRITER_ADVANCE;
begin for I in 1..SIZE-1 loop HEAD:= new BUFFER; HEAD.NEXT:=AVAIL; AVAIL:=HEAD; end loop;
      MARK:= new BUFFER; MARK.NEXT:= null; HEAD:= MARK; TAIL:= MARK;
end QUEUE;
```

Figure 12: The body of a generic queue package, using shared variables.

```
package Q is new QUEUE(T, 4);
task PRODUCER;
task CONSUMER;
task body CONSUMER is
      OK: BOOLEAN;
begin loop Q.READER_ADVANCE(OK);
           if OK then consume(Q.MARK.all); else . . . end if;
       end loop;
end CONSUMER;

task body PRODUCER is
       OK: BOOLEAN;
begin loop Q.WRITER_ADVANCE; produce_data_in(Q.MARK.all); end loop;
end PRODUCER;
```

Figure 13: Tasks communicating via a queue, implemented with shared variables.

Note that the package QUEUE keeps an explicit list of available buffers rather than relying entirely on Ada allocators and Ada's predefined generic unchecked deallocation procedure. Note also that as many buffers as are expected to be needed are preallocated. This is to save time. The general-case storage allocation and deallocation operations supported by the Ada runtime environment are likely to be less efficient, because: (1) arbitrary-sized blocks of storage must be handled; (2) they must be treated as a critical sections, since several tasks may allocate concurrently from a single pool of storage.

Avoiding rendezvous has solved our main problem. That is, neither PRODUCER nor CONSUMER need wait, and both can execute at the same time, if they execute on different physical processors. There remain other problems, such as avoiding starvation and dealing with buffer overflow, but these can be solved independently.

5.4 USING A SERVER TASK

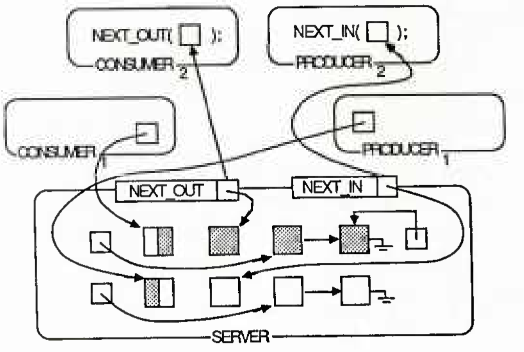
It is unfortunate that the approach taken above does not generalize nicely to more complex task interactions, such as the producer-consumer problem with multiple producers and consumers. It seems that for such problems it may be best to protect shared buffers by interposing a server-task, which controls access to the buffers. Client tasks would then obtain or return buffers by rendezvous with the buffer-server, as shown in Figures 14-16.

```
generic
        type DATA_TYPE is private;
        SIZE: in INTEGER:= 3;
package CHANNEL is
        - - Bounded buffer capable of holding SIZE items, intended for multiple producers and multiple consumers.
        type BUFFER;
        type BUFFER_ACCESS is access BUFFER;
        type BUFFER is record NEXT: BUFFER_ACCESS; DATA: DATA_TYPE; end record;
        task SERVER is
              entry NEXT_IN(A: in out BUFFER_ACCESS);
              entry NEXT_OUT(A: in out BUFFER_ACCESS);
        end SERVER;
end CHANNEL;
```

Figure 14: The specification of a generic channel package, with a server task.

```
package body CHANNEL is
        task body SERVER is
              HEAD, TAIL, AVAIL: BUFFER_ACCESS:= null;
        begin - - Set up list of free buffers.
               for I in 1..SIZE
               loop HEAD:= new BUFFER; HEAD.NEXT:=AVAIL; AVAIL:= HEAD; end loop;
               HEAD:= null;
               - - Begin service.
               loop select accept NEXT_IN(A: in out BUFFER_ACCESS) do
                                  - - Add full buffer, if any, to FIFO.
                                  if A /= null then A.NEXT:= null;
                                        if TAIL= null then HEAD:= A; else TAIL.NEXT:= A; end if;
                                        TAIL:= A;
                                  end if;
                                  - - Get empty item-buffer.
                                  if AVAIL = null then A:=new BUFFER;
                                  else A:= AVAIL; AVAIL:= AVAIL.NEXT; end if;
                           end NEXT_IN;
                      or   accept NEXT_OUT(A: in out BUFFER_ACCESS) do
                                  - - Return empty item-buffer, if any.
                                  if A /= null then A.NEXT:= AVAIL; AVAIL:= A; end if;
                                  - - Get full item-buffer, if any available.
                                  if HEAD /= null then A:=HEAD; HEAD:=HEAD.NEXT; end if;
                           end NEXT_OUT;
                      end select;
               end loop;
        end SERVER;
end CHANNEL;
```

Figure 15: The body of a generic channel package, with a server task.




```
task type CONSUMER;
task type PRODUCER;
PRODUCER_1, PRODUCER_2: PRODUCER;
CONSUMER_1, CONSUMER_2: CONSUMER;
package C is new CHANNEL(T, ACCESS_T, 4);
task body CONSUMER is
    A: C.BUFFER_ACCESS:= null;
begin loop C.SERVER.NEXT_OUT(A);
        if A /= null then consume(A.DATA); else . . . end if;
    end loop;
end CONSUMER;

task body PRODUCER is
    A: C.BUFFER_ACCESS:= null;
begin loop C.SERVER.NEXT_IN(A);
        if A /= null then produce_data_in(A.DATA); else . . . end if;
    end loop;
end PRODUCER;
```

Figure 16: Tasks communicating via channel implemented by server task.

The main problem with this solution is that each datum passed between the producer and the consumer requires not one, but TWO rendezvous. This will probably be time-consuming, but it appears that the only way one can hope to do better is by relying upon implementation-dependent features (e.g., a predefined task type SEMAPHORE, with entries P and V, or an optimization that recognizes certain special forms of tasks).

Given the high cost (in waiting time and overhead) of intertask communication using rendezvous, it seems wise to design so as to keep the need for such communication to a minimum.

5.5 CONTROLLING SCHEDULING

Another one of the problems in using Ada tasks for real-time software is scheduling – that is, controlling the timing of task execution. Ada leaves many of the details of scheduling to the language implementation. It can be partially controlled, however, by means of delay statements and the real-time clock, and by means of timer-generated hardware interrupts linked to entries.

In real-time systems scheduling is typically periodic. The Ada Standard gives the example shown in Figure 17 to illustrate how a delay statement may be used in conjunction with the real-time clock to achieve periodic execution.

```
declare use CALENDAR;
      NEXT_TIME: TIME:= CLOCK + INTERVAL;          -- INTERVAL is a global constant.
begin loop delay NEXT_TIME - CLOCK;
         . . .                                      -- Do some work.
         NEXT_TIME:= NEXT_TIME + INTERVAL;
      end loop;
end;
```

Figure 17: Periodic scheduling with a delay statement.

This will work, but is not likely to be suitable for all applications. Some problems are:

- The delay interval must be large in comparison to the time needed to check the real-time clock, calculate the delay, and execute the delay statement, or the overhead will be excessive.
- The delay statement must be implemented preemptively, and the cycling task must have sufficiently high priority with respect to other tasks competing for the same processor(s) to insure that it can complete an iteration once in each interval.
- The accuracy of timing is limited by the accuracy of the delay statement implementation (even though the variations will average out).

Where more accuracy or a smaller interval is necessary, periodic execution can be achieved by means of a timer-generated hardware interrupt, linked to an entry of a task. The example in Figure 18, adapted from MacLaren [4], illustrates how this technique can be used to program a standard cyclic executive in Ada. In this example, task EXEC calls the several procedures that perform the actual work of the system. EXEC divides time into "minor frames", whose length is the period D of the timer interrupt, and "major frames", whose length is 4D. Procedures INTF and CNTRL are called once every minor frame. Procedure EPROC is called once every 2D seconds, and procedures INTB1 and INTB2 are called once every major frame. The latter are intended to be two parts of a single logical procedure, illustrating how computations that do not fit into a minor frame may be subdivided.

```
task EXEC is
    entry DING;                                              -- indicates the start of a new minor frame
        for DING use at PERIODIC_INTERRUPT'ADDRESS; pragma PRIORITY(10);         -- a high priority
end EXEC;

task body EXEC is
    M: constant INTEGER:= 4;                                 -- number of minor frames in a major frame
    FRAME: INTEGER:= 0;
begin loop accept DING;
        case FRAME is
        when 0 => INTF; INTB1; CNTRL;
        when 1 => INTF; EPROC; CNTRL;
        when 2 => INTF; INTB2; CNTRL;
        when 3 => INTF; EPROC; CNTRL;
        end case;
        FRAME:=(FRAME+1) mod M;
    end loop;
end EXEC;
```

Figure 18: A cyclic executive, implmented using a periodic interrupt entry.

This approach should incur less overhead than using the delay statement, and may be more exact. If there is sufficient processor time left, it permits other (background) tasks to run, at lower priorities. There are also some limitations, however:

- Any other interrupt handlers must be kept sufficiently short and their interrupts sufficiently infrequent that the computations to be performed in each minor cycle can be completed, in the worst case.
- All the work to be scheduled by the executive must be broken into procedures that can be executed within one minor frame.
- A task that must respond to interrupts cannot wait for other tasks, which makes it difficult to pass information between interrupt-driven tasks and other tasks, as explained above.
- This solution is based on a sequential, single-processor model of execution. (It can be generalized, however, using trivial entry calls rather than procedure calls, to work with a parallel model of execution.)
- There is likely to be disagreement among compilers on details of the implementation of interrupt entries. For example: What happens when a task with an interrupt entry is itself interrupted? Does this depend on whether the task is executing an accept on an interrupt entry at the time? Also, what happens if the task responsible for handling an interrrupt is not waiting at an accept for the interrupt when the interrupt arrives? (In particular, what happens if the task EXEC above is still executing in the case statement when the next time interrupt arrives?) Can an interrupt-driven task rendezvous with other tasks? What happens if an interrupt occurs during such a rendezvous?

5.6 STARTING UP AND SHUTTING DOWN

Starting up a task and shutting it down are both time-consuming operations, and can involve subtle interactions with other tasks.

Before a task can start up it must be created. This may be as a consequence of the elaboration of a task declaration, or elaboration of an object declaration or allocator for a task type or a composite type containing a task component. Elaboration typically involves the execution of code produced by the compiler, and is performed each time execution enters the scope of the declaration. This minimally involves allocating storage for the task, and initializing data structures used by the runtime environment to coordinate task execution.

At some point after it is created, a task begins activation. This involves the elaboration of the declarative part of the task, including the initialization of local data objects, and the creation and activation of any local tasks. The time it takes to execute the elaboration code of a task thus depends on the complexity of its local declarations. Eventually, when a task has finished activation, execution of the sequence of statements of its body may begin.

There are a number of fairly complex rules governing task activation. These rules help to prevent execution of a task before all of the data structures on which it depends have been initialized, and calls to an entry of a task that has not yet been created. Since there is still danger of such elaboration order errors, Ada programs also check for them at run-time. For example, consider the following pair of packages:

```
package P is task type T; end P;
package Q is T1: T; end Q;
package body Q is . . .
        - - Task T1 is activated here, just before the "begin".
        - - ELABORATION_CHECK then raises PROGRAM_ERROR, because the body of T has not yet been elaborated.
begin . . . end Q;
package body P is
        . . .                                                        - - declarations used by task body T
        task body T is . . . end T;
end P;
```

Figure 19: A problem with task body elaboration order.

Task T1 is declared in package Q, whose body is elaborated before the body of T, T1's task type. The elaboration of Q's body calls for T1 to be activated, and this raises the exception PROGRAM_ERROR. In this example, the error is obvious – the order of the two package bodies should be reversed – but problems with elaboration order are not always so obvious, nor so easily corrected. It is best to avoid this kind of problem by paying careful attention to activation order in the design of a system.

Should there be need to start up a collection of tasks synchronously, after each completes its elaboration it can wait for a "go ahead" signal, except for one, which serves as starter. The starter can give the signal by calling a designated entry of each other task, in a specified sequence. Such entry calls can also be used to "restart" tasks in response to some event, though this requires that the restartable tasks be designed so they are always willing to accept such a call.

Execution of a task may continue indefinitely, or it may complete. Completion may be a result of: the task reaching the end of its sequence of statements; an exception that is raised and not handled within the task; the task being aborted; an orderly shutdown of a family of inactive tasks, using the "terminate alternative" of a selective wait statement.

Some time after a task completes execution, it may be terminated. The distinction between completion and termination is similar to the distinction between activation and creation. As with activation, there are a number of fairly complex rules governing termination. Their effect is to insure that a task continues to exist so long as there is another active task that may attempt to interact with it, and that all the data that a task may access must continue to exist so long as the task remains active. For example, in the following example, the procedure P is not permitted to return, because the task T never completes, even though P itself has no statements.

```
procedure P is
        task T;
        task body T is begin loop null; end loop; end T;.
begin null; end P;
```

Figure 20: A master of tasks may not be exited until all dependent tasks have terminated.

Because task creation, activation, and termination are fairly complex operations, they can contribute significantly to the execution time of a program. Moreover, if tasks are routinely created and destroyed during the normal operation of a system, the execution timing is likely to be erratic. It is therefore preferable in the design of time-critical systems that all tasks be created and activated before the system becomes operational, during "warm-up" time.

When one task in a collection of communicating tasks does die unexpectedly, perhaps due to an error, it may be preferable that the entire collection die and be recreated. This is likely to be time-consuming, but offers the only chance of complete recovery. In the example given below, a watchdog task aborts an entire collection of tasks if an error is detected. The enclosing loop ensures that they will be restarted.

```
loop declare
    task WATCHDOG;
    task body WATCHDOG is
        task A . . . ; task B . . . ; . . .
    begin loop . . .
            if FAILURE_DETECTED then abort; end if;                   - - (1)
         end loop;
    exception when others => abort;                                   - - (2)
    end WATCHDOG;
    begin null;
    exception when others => null;                                    - - (3)
    end;
end loop;
```

Figure 21: Restarting a collection of tasks.

Note that the execution of the task enclosing the outer loop cannot leave the block statement until all the tasks within have terminated. This will not happen until the watchdog dies. Note also that all the other tasks (including A, B) are made dependents of the watchdog so that when it aborts itself (1) it aborts all of them, without needing to know their names. Thus, even tasks designated by access types declared within the watchdog will be aborted. The inner exception handler (2) insures that all tasks will be aborted, whether the failure is detected directly, or through an exception being raised within the watchdog task. The outer exception handler (3) insures that the tasks will be restarted even if an exception is raised during the activation of the watchdog.

6 THE RADAR TRACKING EXAMPLE

Let us now consider in more detail how we might implement a solution to the radar tracking problem outlined above, using Ada tasks. The work which the software must do can be divided as shown below. The information flow between these tasks is as shown in Figure 2.

```
(1) task COMMAND_PROCESSOR;
        -- Respond to console interrupts, accumulate characters, and when a
        -- full command is accumulated, execute it within C seconds.

(2) task DISPLAYER;
        -- Every D seconds, update the screen image, using current data from the track file.

(3) task RADAR_DISPATCHER;
        -- Every R seconds, check the radar, and feed it another order, if necessary.

(4) task RADAR_RECEIVER;
        -- Respond to interrupts from the radar when it has filled up a DMA
        -- buffer area, informing it of the next buffer area to use.
(5) task BACKGROUND_SEARCHER;
        -- Continually generate radar orders to search for new objects.
(4) task TRACKER;
        -- Every T seconds, update the track file, using new radar data, if possible.
        -- Attempt to obtain new radar data for each tracked object every T seconds.
```

The partitioning above is based on the principle of initially not introducing any more task divisions than seem necessary to reflect the inherent asynchronisms of the problem. The Command Processor, the Radar Dispatcher, and the Radar Receiver are driven at different asynchronous rates by their respective data streams. The Displayer and the Tracker are also driven at different rates, according to the required timeliness of their outputs. The Background Searcher needs to operate independently, so that it can use up otherwise unneeded radar and processor capacity.

The tasks in our example all have the characteristic that they will only need to be created once, when the system is started up, and should not terminate during the normal operation of the system. They could all be declared in the declarative part of the main program or a library package referenced by the main program.

6.1 THE COMMAND PROCESSOR

The Command Processor must be able to respond quickly to operator keystrokes, so that no data is lost. At the same time, it must synchronize with other tasks, in order to execute operator commands. These two requirements are incompatible. We therefore introduce a new task, the Console Monitor, which will be responsible for responding to console interrupts, accumulating characters, and passing on complete commands to the Command Processor. There are several ways the Console Monitor and its communication with the Command Processor can be implemented, depending on the sophistication of the compiler. We will consider two ways.

With a primitive compiler, like the author's FSU/AFATL compiler, first-level interrupt handlers must be coded in assembly language. Such an interrupt handler, coded in assembly language, would in effect be the console-monitoring "task". It would buffer up to one full command, and then pass it on to the Command Processor via a "mailbox" – a dummy task control block which can be linked to an interrupt entry queue and which simulates a an entry call. Since many such mailboxes can be linked to any entry queue, there is no need for the handler to wait, nor is there danger of lost commands. This interface is shown in Figure 22.

```
task COMMAND_PROCESSOR is
    entry RECEIVE_COMMAND(C: in COMMAND);
        for RECEIVE_COMMAND use at CONSOLE_HANDLER'ADDRESS;
end COMMAND_PROCESSOR;

task body COMMAND_PROCESSOR is
begin loop accept RECEIVE_COMMAND(C: in COMMAND) do
            ...                                            -- Save C;
        end RECEIVE_COMMAND;
        ...                                                -- Execute command C;
    end loop;
end COMMAND_PROCESSOR;
```

Figure 22: The Command Processor task.

Alternatively, the compiler may permit coding the first-level interrupt handler directly in Ada. For example, the ALS compiler, developed by Softech, Inc., provides special-case treatment of "fast interrupts", "trivial entries", and "trivial accepts"[5]. With such a compiler, the console-monitoring could be done by an Ada task, as illustrated in Figure 23. . This task uses a trivial entry to signal the Command Processor when it haws accumulated a complete command. Trivial entry calls are guaranteed to be accepted without delay, but may not have parameters. For this reason, the command must be passed in a shared-variable buffer. Exchange-buffering should probably be used, but such details are omitted.

```
task CONSOLE_MONITOR is
    pragma INTERRUPT_HANDLER_TASK;
    entry KEYSTROKE;
        for KEYSTROKE use at CONSOLE_INTERRUPT'ADDRESS;
        pragma FAST_INTERRUPT_ENTRY(KEYSTROKE, SIGNALLING_QUICK);
end CONSOLE_MONITOR;

task COMMAND_PROCESSOR is
    entry SIGNAL; pragma TRIVIAL_ENTRY(SIGNAL);
end COMMAND_PROCESSOR;

task body CONSOLE_MONITOR is
begin loop accept KEYSTROKE do; . . .          -- Move character from input port to buffer;
            if . . .                           -- If the buffer contains a complete command
            then . . .                         -- exchange buffers
                select COMMAND_PROCESSOR.SIGNAL;  -- Conditional entry call guarantees no waiting.
                else null;
                end select;
            end if;
        end KEYSTROKE;
    end loop;
end CONSOLE_MONITOR;

task body COMMAND_PROCESSOR is
begin loop accept SIGNAL;                      -- A trivial accept statement.
        . . .                                  -- Process the command;
    end loop;
end COMMAND_PROCESSOR;
```

Figure 23: An alternate version of the Command Processor task.

We still need to insure that a command is processed within C seconds. One way is to give the Command Processor very high priority, so that it need never wait for any other task; then care must be taken in the programming of the Command Processor that the processing time for each command is short enough not to interfere with the timely execution of other tasks. Other solutions require more detailed consideration of the other tasks within the system and the hardware configuration. We will consider some of these other approaches for the Displayer, next.

6.2 THE DISPLAYER

The Displayer must update the display once every D seconds. We assume that D is fairly large (say one or two seconds), and that some variation in the period is allowable, so that we can schedule the execution of this task with a delay statement. There still may be a problem, though. Updating the display may take longer than we can afford to wait between execution of other tasks. One way to make sure that the Displayer does not prevent other tasks from executing is to put it on a separate processor. Otherwise, we could try giving it a fairly low priority, but then when the system overloads the Displayer may not execute at all.

One idea is to set a deadline for the Displayer, and raise its priority when it nears the deadline. Since Ada priorities are static, this would be impossible with a standard runtime environment. (Note, however, that an implementation conforming to the Standard may provide dynamic priorities, restricting the standard static priorities to a null range.) Nevertheless, the desired behaviour can be achieved, though awkwardly. The first step is to break the work done by the Displayer into small units, that we can afford to execute without switching tasks; that is, into the same kind of divisions required for a cyclic executive.

Suppose we code these small units of work as cases within a single procedure, WORK, which looks at a global variable, NEXT_STAGE, to see how far work has progressed, and which stage it should execute at each call. Just before returning, procedure WORK increments NEXT_STAGE. Also, when it finishes updating the display, it resets NEXT_STAGE to zero, LAST_TIME to the time it finished updating the display, NEXT_TIME to the time it is due to start the next update, at low priority, and DEADLINE to a time after which updating the display must be given high priority. We encapsulate the code relating to the Displayer's logic in the following package:

```
package DISPLAYER is
        LAST_TIME, NEXT_TIME, DEADLINE: TIME;
                pragma SHARED(LAST_TIME);
                pragma SHARED(NEXT_TIME);
                pragma SHARED(DEADLINE);
        procedure WORK;
end DISPLAYER;

package body DISPLAYER is
        NEXT_STAGE: INTEGER:= 0;
        procedure WORK is . . . ;
begin . . .                    -- Initialize LAST_TIME, NEXT_TIME, DEADLINE.
end DISPLAYER;
```

Figure 24: The Displayer.

Given this breakdown, there are several ways we can proceed. If we already have a cyclic executive, we can intersperse calls to WORK within its schedule. Alternatively, we might create a new high-priority task, DISPLAY_EXEC, which alternates between executing WORK and delaying itself. This task can use the values of NEXT_STAGE, LAST_TIME, and the real-time clock to adjust the duration of its delay as the deadline for updating the display approaches.

```
task DISPLAY_EXEC is
    pragma PRIORITY(10);
end DISPLAY_EXEC;

task body DISPLAY_EXEC is
    . . .
begin loop delay(DISPLAY.NEXT_TIME - CLOCK);
        T:= DISPLAY.LAST_TIME;
        while DISPLAY.LAST_TIME = T loop
            DISPLAY.WORK;
            if CLOCK < DISPLAY.DEADLINE then delay . . . ; end if;
        end loop;
    end loop;
end DISPLAY_EXEC;
```

Figure 25: The Displayer Executive.

Of course, care must be taken that the units of work and the delay durations are sufficiently large that the overhead (of the delay statement, computing the duration, and the calls to procedure WORK) is tolerable. There is also a danger, present any time there are several tasks with delay statements, that several tasks will get in phase with one another. If this happens, the processing resources of the system will alternate between idleness and overload, with the the overloads possibly escalating until there is complete system failure.

We can solve the problem of DISPLAY_EXEC delaying itself longer than necessary by means of another task of lower priority. Ideally, the lower priority task would execute until the deadline is passed, making use of any slack time, then when the deadline passes the higher priority task would take over. A difficulty with this approach is ensuring that at most one of the two tasks is executing WORK at any one time. The obvious solution is rendezvous, if the execution time of one call to WORK is long enough to justify the cost of the rendezvous. Another difficulty is that if the deadline passes while the lower priority task is executing WORK the higher priority task must wait for it to complete. This can be solved by letting the higher priority task do all the work, with the lower priority task merely serving as a gauge of system workload.

```
task SLOW_EXEC is pragma PRIORITY(5); end SLOW_EXEC;
    . . .
task body DISPLAY_EXEC is
    . . .
    if CLOCK < DISPLAY.DEADLINE
    then select accept PROD; or delay X; end select; end if;
    . . .
end DISPLAY_EXEC;

task body SLOW_EXEC is
begin loop DISPLAY_EXEC.PROD;
    end loop;
end SLOW_EXEC;
```

Figure 26: A two-level executive.

Note that we have been forced to reject another approach that seems good, but which Ada does not support. This would be to have the lower priority task do the work, but let the higher priority task give it a boost periodically, by calling it whenever the deadline is passed. The body of the higher priority task might be structured as shown in Figure 27.

```
task body BOOSTER is                                                    - - a very high priority task.
begin loop delay(DISPLAYER.DEADLINE-CLOCK);
          if DISPLAY.LAST_TIME ¡ DISPLAY.DEADLINE then DISPLAY_EXEC.BOOST; end if;
     end loop;
end BOOSTER;
```

Figure 27: An approach that will not work.

The effective priority of DISPLAY_EXEC would be raised to that of BOOSTER while it is accepting a call from BOOSTER. The problem, of course, is that the priority is not boosted until the call is accepted, and the accepting task will not be able to reach an accept statement if its priority is too low to begin with. Alas, this exposes a fundamental inconsistency in the Ada semantics of priorities. (For consistency, since the effective priority of a task can be raised by rendezvous with a higher priority task, we would expect that it should also be raised by the presence of a high-priority task on one of its entry queues, and that the order in which calls are accepted should be FIFO within priority classes, rather than strict FIFO, but this is not true.)

A third approach might be to redesign WORK so that more than one call could execute concurrently, without conflict. This would require partitioning the data structures updated by WORK so that those accessed in different stages are disjoint. We consider this approach in more detail below, for the Tracker.

Beneath all of these examples lurks an important truth: achieving satisfactory performance from a real-time system is likely to require "tuning" the task-level structure of the system. Several different task structures may need to be tried before one is found that is satisfactory. Moreover, subsequent "maintenance" may change timing characteristics of some components sufficiently that this tuning must be redone.

If such task-level design changes are to be possible without extensive recoding, most of a system's code will need to be organized into procedures. For maximum scheduling flexibility, the procedures out of which tasks are built should be short enough to be executed without preemption (excluding preemptions to service hardware interrupts), yet long enough so that the overhead of a task-switch between calls is tolerable.

6.3 THE RADAR DISPATCHER

The Radar Dispatcher task poses problems similar to those of the console Command Processor, though the rate of data flow is much higher. We assume the radar is speedy, so that the period, R, with which it must be polled is too small to make use of a delay statement practical. It will therefore be driven by a periodic interrupt. On the other hand, it must be able to receive orders from both the Background Searcher and the Tracker. These tasks could pass the orders to the Radar Dispatcher via entry calls. However, if the overhead of a delay is too high, then the overhead of an entry call probably is too high also. A solution is to deliver the orders to the Radar Dispatcher in batches, so as to spread out the cost of the rendezvous. As with the console-monitoring task, we have the choice of implementing the radar polling below the level of Ada, in assembly language, or using a fast-interrupt entry if the compiler supports it. The Radar Poller will take its orders from a buffer. For buffering we can use the circular buffering scheme given in the producer-consumer example above, which does not require rendezvous. However, this permits only one producer and one consumer. In this case there is only one consumer, the Radar Poller, but there are two producers. In order to accomodate them both, we interpose the Radar Dispatcher. The Background Searcher and the Tracker will send their orders to the Radar Dispatcher in batches, and the radar dispatcher will channel them to the Radar Poller. This arrangement not only solves the problem of multiple access, but gives us an opportunity to control the flow of orders, and perhaps to sort them. (For example, we may wish to give preference to orders from the Tracker.) It also limits the scope of software changes if we decide to add or change radar units. We end up with the organization shown in Figures 28 and 29.

6.4 THE RADAR RECEIVER

The Radar Receiver is also interrupt-driven. Its main function is to channel reports to the Tracker. So long as there is only one Tracker task, there is no reason to bother with rendezvous; we can use a shared-variable buffer to pass data from the interrupt-handler to the Tracker directly. This is shown in Figure 30.

6.5 THE BACKGROUND SEARCHER

The job of the Background Searcher is to look for new objects, not already in track. It could be treated as a pure background task, executing at the lowest possible priority. Alternatively, we may wish to guarantee a certain minimum level of performance, even when the system is heavily loaded, tracking many objects. If so, we can treat this task similarly to the Displayer, breaking it into procedures of moderate length. This would have an advantage in flexibility. For example, we might wish the operator to be able to modify the effective priority of the search, as well as to direct its pattern.

If the Background Searcher is implemented as a low priority task we might be able to afford aborting one version and starting up another. We might even create additional background search tasks, in response to operator commands, so as to do concentrated searches in special regions.

```
task RADAR_POLLER is
    pragma INTERRUPT_HANDLER_TASK;
    entry DING;
        for DING use at PERIODIC_INTERRUPT'ADDRESS;
        pragma FAST_INTERRUPT_ENTRY(DING, SIMPLE_QUICK);end RADAR_POLLER;

task RADAR_DISPATCHER is
    entry PROD; pragma TRIVIAL_ENTRY(PROD);
    entry TRACKER_ORDER(B: in out COMMAND_BUFFER_ACCESS);
    entry BACKGROUND_ORDER(B: in out COMMAND_BUFFER_ACCESS);
    end RADAR_DISPATCHER;

    task body RADAR_POLLER is
    begin loop accept DING;
            if RADAR_IN.STATUS.READY and SOME_ORDERS_HERE
            then . . .                    - - Take the next command from the buffer and send it to the radar.
            end if;
            if FEW_ORDERS_HERE then RADAR_DISPATCHER.PROD; end if;
        end loop;
    end RADAR_POLLER;
```

Figure 28: The specifications of the Radar Dispatcher and Poller tasks.

```
task body RADAR_DISPATCHER is
begin loop select accept TRACKER_ORDER(B: in out COMMAND_BUFFER_ACCESS) do
                . . .
            end TRACKER_ORDER;
    or  when FEW_ORDERS_TO_DISPATCH =>
            - - Make the Background Search wait until the radar load is not too heavy.
            accept BACKGROUND_ORDER(B: in out COMMAND_BUFFER_ACCESS) do
                . . .
            end BACKGROUND_ORDER;
    or  when SOME_ORDERS_TO_DISPATCH =>
            accept PROD; . . .            - - Transfer orders from Dispatcher buffer to Poller's buffer.
        or  delay . . . ;
        . . .                             - - Check on the radar, to see if it is working.
        end select;
    end loop;
end RADAR_DISPATCHER;
```

Figure 29: The bodies of the Radar Dispatcher and Poller tasks.

```
task RADAR_RECEIVER is
    pragma INTERRUPT_HANDLER_TASK;
    entry FRESH_DATA;
        for FRESH_DATA use RADAR_SIGNAL'ADDRESS;
        pragma FAST_INTERRUPT_ENTRY(FRESH_DATA,SIMPLE_QUICK);
end RADAR_RECEIVER;

task body RADAR_RECEIVER is
begin loop accept FRESH_DATA do
            . . .                         - - Tell radar where to put next batch of reports.
        end FRESH_DATA;
    end loop;
end RADAR_RECEIVER;
```

Figure 30: The Radar Receiver task.

6.6 THE TRACKER

The Tracker incorporates most of the system's intelligence. Without going into details of the Tracker's algorithm, we see that its work divides naturally into two phases, one driven by the structure of the track file and the other driven by the order of the incoming radar reports.

First, because the estimated position of each tracked object is to be accurate within a tolerance of T seconds, the Tracker will need to update the position of all objects once every T seconds. It will update the position of each object using radar data, if any is available; otherwise, it will extrapolate the position of the object based on its previous position and estimated velocity. After several cycles without radar confirmation, the object will be considered lost, and deleted from the track file. (Special measures might be taken first to bring the object back into track, but we ignore these details here.) At the same time as it updates the position of an object, the Tracker can generate a radar order to obtain the data it will need to update the position of that object in the next cycle. All of this processing can be viewed as being performed for each object during a pass over the track file, one pass being made every T seconds.

Second, the Tracker needs to correlate incoming radar information with items in the track file, based on spacial proximity. The track file may be organized so as to help in doing this efficiently. Since the radar reports will be coming in in arbitrary order, and may include reports of new objects, not yet in the track file, the Tracker must set up a correspondence between radar reports and items in the track file before it can use the reports to update positions. This may be done during periodic passes though the recently-arrived radar reports, by sorting them out according to the organization of the track file. There will be computations that need to be performed on a per-report basis, including coordinate conversion and interpretation of multiobject reports, that may also be performed during this phase.

If there is only one processor, the Tracker can be organized as two phases within one task. For example, see Figure 31.

```
task body TRACKER is
begin loop START_TIME:= CLOCK;                          -- Phase I: (per-report)
           Swap_radar_report_buffers;
           for I in 1..LAST_REPORT_IN_BUFFER loop
           . . .                                        -- process Ith report.
           end loop;
                                                        -- Phase II: (per-track)
           for I in 1..LAST_TRACK loop
           . . .                                        -- process Ith track.
           end loop;
           COMPLETION_TIME:= CLOCK;
           delay T - (COMPLETION_TIME - START_TIME);
      end loop;
end TRACKER;
```

Figure 31: The Tracker task.

If the radar is speedy and there are many objects to track, this is likely to be more work than a single processor could handle. The hardware solution is to use more processors. The software problem is how to use them. We will return to this problem after we have handled another – controlling access by the different tasks to the track file and radar reports.

6.7 THE TRACK FILE

The track file presents a problem of shared access, since it must be read by the Displayer and both read and updated by the Tracker and the Command Processor. In addition, other software (e.g. a collision-detector) may need to read from it. Thus, some means of mutual exclusion must be provided.

Since several different tasks will need to access the track file, and because we may want to add more tasks later, we are forced to view it as a client-server problem, where the tasks accessing the track file do so by obtaining permission from a Track File Server task. In order that client tasks do not have to make rendezvous with this server too often, it should allocate access in fairly large blocks. Since we cannot afford time to copy such large blocks of data frequently, the server must provide access to data either through access values (pointers) or through assigned index ranges in a shared array. To make such large blocks of data useful, we may partition the data in the track file into blocks representing different contiguous sectors of the physical space in which objects are to be tracked.

Since we cannot yet predict the order in which all the tasks will need to access this data, we must consider the possibilities of deadlock and starvation. To lessen this danger, we will prohibit in-place updating of data. That is, a task such as the Displayer, that requests read-only access to a sector of the track file will receive access to the most recent version of that sector in the database. A task, such as the Tracker, that updates the track file must do so by requesting read-access to the database, and permission to update. The server grants read-only access to the current copy of the sector, and update access to an available block of memory, which will become the new version of that sector when returned to the database. Thus, simultaneously one task may be updating and several tasks may be reading from the same sector of the track file, but no more than one may update at the same time. Time stamps are applied to the sectors of the track file, since the versions of the different sectors are not synchronized.

We thus have the organization shown in Figure 32.

```
task TRACK_FILE_SERVER is
    entry REQUEST_READ(SECTOR_INDEX)(S: out SECTOR_ACCESS);
    entry RELEASE_READ(A: in SECTOR_ACCESS);
    entry START_UPDATE(SECTOR_INDEX)(S: out SECTOR_ACCESS);
    entry FINISH_UPDATE(SECTOR_INDEX);
end TRACK_FILE_SERVER;

task type TRACKER;
task body TRACKER is
begin loop . . .                                    -- Choose a sector, I, to update.
        TRACK_FILE_SERVER.START_UPDATE(I)(S);
        . . .                                       -- Perform update of S.all.
        TRACK_FILE_SERVER.FINISH_UPDATE(I);
    end loop;
end TRACKER;
```

Figure 32: The Track-file Server task.

6.8 BUFFERING RADAR REPORTS

We have a similar, but less extreme, problem with the storage of radar reports between the time they are received and the time they are processed by the Tracker. In order that the track file can be updated a sector at a time, we want the Radar Receiver to sort the incoming radar reports into different "bins" according to sector. The Tracker will thus simultaneously need read access to a bin of radar reports and update access to the corresponding sector of the track file. Anticipating deadlock problems if we later divide the Tracker into several tasks executed on different processors, we plan to make the allocation of report bins between Tracker processes implicit, through the allocation of update rights to track-file segments.

We still have the problem of providing write-access for the Radar Receiver to the report bins. When the Tracker requests update access to a sector, we want it to take all the most up-to-date radar reports for that sector. On the other hand, we do not want the Tracker to rendezvous with the Radar Receiver for each incoming radar report, because the overhead would be excessive.

One solution is to used shared variables for mutual exclusion, representing each bin as a linked chain of buffers. In particular, suppose we use the generic package QUEUE, defined above, with one instantiation per sector of the track file. A troublesome detail is that Ada only permits us to specify the pragma SHARED for simple variables – not for components of an array or record. This forces us to treat each queue as a package (rather than as a record, which would have allowed the bins to be represented as an array of queues, indexed by sector identifier). Instead, the reader and writer will have to use case statements to select the bin for a given sector.

6.9 USING PHYSICAL PARALLELISM

Suppose the radar tracking system is to be implemented on a multiprocessor configuration, like that shown in Figure 3, above. Using present-day compilers, the best solution would be to write a separate Ada program for each processor, using conventional locking techniques (e.g. test-and-set) to protect shared data.

At some time in the future it is likely that there will be Ada compilers that are capable of distributing a single program over such an architecture. Though radical approaches have been proposed[2], a simple and relatively easy first step toward a multiprocessor implementation seems to be compiling a single Ada program for a single virtual memory address space, with shared access by a collection of processors. Virtually no change is required from a compiler for a single processor. Some changes would need to be made in the runtime support software, but they would not need to be major. Each processor would perform its own dispatching, working from a common queue of ready tasks. No special pragmas or language extensions would be necessary, since the allocation of processors to tasks could be completely dynamic. Pragmas could be provided, however, to insure that certain processors are reserved for critical tasks.

Let us assume that such a multiprocessor compiler is available, and consider how the tasks of our tracking system might be distributed over the processors in Figure 3. We might assign the Console Monitor and Command Processor tasks to Processor C, and the radar interface tasks to Processor D, leaving Processors A and B, and any slack time on processors C and D, available for the Tracker and Background Search.

Since the Tracker task seems to be the most demanding of processor time, it would be helpful to split its work between Processors A and B. Without giving up the two-phase structure, we can separate Phases I and II, so that they execute as a pipeline. While Processor B is doing Phase II, updating the track file, Processor A can do the Phase I processing for a collection of radar reports that will be used by Processor B the next time it updates the track file. This is not an ideal solution, however. One problem is that the excess time of Processors C and D is not used, except perhaps for the Background Search. Another problem is that the pass-oriented structure is likely to put an uneven demand on the radar, resulting in a need for long queues. Finally, radar data received immediately after the start of an update pass cannot be used until the next pass, by which time is is "stale".

Alternatively, we might split the Tracker "vertically", instead of "horizontally". We could create several objects of task type Tracker, so that one Tracker could be updating one sector of the track file on Processor A while the other is updating another sector on Processor B.

To obtain maximum speed from parallel processors, data shared between different processors should be divided between different memory modules, so that processors do not waste time contending for memory access. Thus, for example, the even numbered track file sectors and their corresponding report bins might be assigned to Memory 0, and the odd ones to Memory 1. A copy of the Tracker code might be resident in each of Memories 2 and 3. We might have two Tracker tasks, one with its workspace assigned to Memory 0 and the other with its workspace asigned to Memory 1. The code for the Console and Background Search Tasks could go into Memory 4, and their workspaces into Memory 5. Memories 6 and 7 would be used for the Radar tasks.

Of course, this solution is still not very good, since it is likely that most of the tracking activity may be concentrated in one sector. This suggests that a more subtle approach may be needed if we were to design a real tracking system.

Note that we have ignored one seemingly elegant approach – using one Tracker task per tracked object. This was considered and rejected, because of several problems, including especially:
(1) heavy execution time overhead, due to extra rendezvous and individual dispatching of objects;
(2) storage overhead, due to extra task control information.

7 CONCLUSIONS

We have described some of the features of Ada that support programming with tasks, and considered some of the ways they might be used in real-time embedded software systems. There are still many interesting features which we have not covered, however. These include:

- interactions storage management, including the STORAGE_SIZE clause for task types;
- interactions with exception handling, including exceptions raised by task operations, propagation of exceptions between tasks, and response of tasks to exceptions;
- entry families (i.e., "arrays" of entries);
- aborting a task;
- details of activation;
- details of completion and termination, including the terminate alternative;
- the several forms of selective wait statement;
- peculiarities of main programs and of tasks declared in library packages;
- attributes 'CALLABLE, 'TERMINATED, and 'COUNT.

The full definition of the semantics of Ada tasks is in [1], and a discussion of some implementation techniques is given in [6].

Though Ada may not be ideally suited for real-time software, especially from the point of view of obtaining maximum performance from parallel hardware architectures, it can be made to work. Moreover, because Ada does provide linguistic means for describing concurrency, it is a vast improvement over languages that do not.

In order to support the development of good real-time systems, compilers and runtime environments will need to provide more than the minimal features required by the Ada Standard. At the same time, to permit efficient mapping onto hardware, they may need to impose limitations on the use of some standard language features. (A good example is fast interrupt entries.) So long as dependencies on such implementation features are explicit, and isolated to a small number of code modules, the benefits will outweigh the costs. Indeed, with real-time systems some implementation-dependent programming seems unavoidable.

We do believe, however, that there is serious cause for concern about the complexity of Ada compilers, and the effect that this is likely to have on their reliability. This complexity appears likely to be aggravated by optimizations that implement a single language construct in different ways, depending on context. This author hopes that the Department of Defense may eventually recognize the wisdom of designating a restricted subset of Ada, which can be implemented efficiently without need for such special-casing.

Finally, as a positive step toward using Ada to develop real-time systems that can be flexibly tuned to meet timing constraints and fitted to peculiar hardware configurations, we would like to suggest an approach to structuring Ada programs. This is based on an adaption of a proven methodology [7], which recognizes three kinds of components – (1) procedural components, (2) informational components, and (3) structural components.

Procedural components do the "work" of the system, by transforming data. These can be realized by Ada procedures and functions. Ideally, the primary (top level) procedural components should have the following properties:

1. They should be short enough that they can be executed to completion without causing scheduling problems, but long enough that making a rendezvous or otherwise switching a processor from one task to another between them would not incur disproportionate overhead.
2. They should access data only through explicit parameters.
3. They should not depend in any way on the order in which they are called, how they may be called by different tasks, or what processor happens to execute them.

Informational components hold data between transformations. These can be realized by Ada packages containing (mainly) object declarations. Ideally, these should be partitioned in such a way as to avoid need for synchronization between concurrent operations.

Organizational components describe which transformations should be applied to which data, in what order, and at what times. These can be realized by combinations of of Ada tasks. They should be designed to take into account the actual hardware resources and costs of intertask communication. They are likely to need to be reorganized to tune the entire system's performance.

In addition, for Ada we would add two more kinds of components, to take advantage of Ada's strengths in the area of abstraction, extendability, and reusability – (4) data-type modules and (5) communication modules. Data-type modules consist of packages containing declarations of data types and operations on those types whose usefulness goes beyond one particular program (i.e. subprograms at a lower level than primary procedural components, and that have the character of language extensions). Communication modules provide abstract communication interfaces for tasks or main programs, hiding lower-level protocols and in some cases even the identities of the communicating entitites. (For example, consider MASCOT channels and pools.) Both kinds of modules should typically be suitable for expression as generic packages.

References

[1] United States Department of Defense, Military Standard Ada Programming Language, ANSI/MIL-STD-1815A Document , U.S. Government Printing Office, Washington, D.C. (January 1983) paragraphs 9.0.1-9.0.5.

[2] D. Cornhill, J. Beane, and J. Silverman, "Distributed Ada Project - 1982", technical report, Honeywell, Minneapolis, MN (January, 1983).

[3] A.N. Haberman and I.R. Nassi, "Efficient implementation of Ada tasks", technical report, Department of Computer Science, Carnegie-Mellon University (January 1980).

[4] L. MacLaren, "Evolving toward Ada in real time systems", Proceedings of the ACM SIGPLAN Conference on Ada, Boston (1980) 146-155.

[5] W.R. Greene and J. Juergens, "Fast interrupt entries", paper presented at AdaTEC/AdaJUG Meeting, San Jose (February 1985).

[6] T.P. Baker and G.A. Riccardi, "Ada tasking: from semantics to efficient implementation", IEEE Software 2.2 (March 1985) 34-45.

[7] T.P. Baker and G. Scallon, "An architecture for real-time software systems", University of Washington Department of Computer Science technical report 85-06-04, to appear in IEEE Software .

REUSABLE SOFTWARE

Olivier ROUBINE
INFORMATIQUE INTERNATIONALE
Centre de Développement de Sophia-Antipolis
Les Cardoulines - Route des Dolines
06560 VALBONNE - FRANCE

ABSTRACT

The design of the language Ada® was undertaken in an effort to reduce the cost of software. One of the overriding factors to achieve such a benefit is the possibility to reuse program components among different projects, leading to a true software component industry. The various aspects of Ada that relate to this notion are discussed, and in particular the relevance of packages, separate compilation, generic units, abstraction and portability. Matters relating to the production, distribution and use of software components are also discussed.

1. INTRODUCTION

The development of Ada had as its primary objective a reduction of soaring software costs. Such a reduction could be obtained through a combination of various factors:

- a strongly-typed, high-level language allowing numerous errors to be detected by the compiler;
- a unique language, usable for all mission-critical computer applications, that would reduce development costs (no duplication of tools and environments) and training costs (programmers using the same language from one project to the other);
- a language that woud support the development of systems by building up on existing software.

This latter aspect is probably the most prominent one, because its benefits can be easily quantified.

In the past, software reuse has been hindered by the fact that it was quite difficult to isolate program parts that could be reused in the context of another project (a noticeable exception is the notion of libraries of subprograms, such as scientific or graphic libraries; however, such libraries generally offer only very basic components).

We will see how Ada addresses the issue of software reuse through various language features that not only support the reuse of software components, but also promotes the development of software so as to isolate reusable components.

The paper first attempts to give a characterization of what constitutes a good component; it then discusses the economics of reusable software, and the general problems involved. It then reviews the various features of Ada that are relevant to the development and use of software components. Lastly, it presents an investigation of the methodological implications of reusable components, i.e., how to design systems that use components, and how to construct systems so that part of them can be turned into reusable components.

2. CHARACTERIZATION OF REUSABLE SOFTWARE

The central notion of reusable software is the reuse of a complete software system, or parts of it, in a different context than the one it was originally developed for.

Without this latter qualification, we are confronted with a much more conventional problem that of portable software. We will see that portability is an important property of reusable software but it is not the only one.

Although the reuse of complete programs is not uncommon, e.g., in the context of software development environments where primitive software tools are often composed to make more complex tools, it is rarely the case that complete embedded computer programs can be reused in a different context.

Embedded systems tend to be rather complex and specific. Yet they still contain many opportunities to reuse software that has been previously developed in a different context. Some examples are given below:

a/ Track Processing in Radar Systems
In a radar system, echos must be processed to recognize consecutive echos of the same object, and to record various data associated to a track, (speed, coordinates, etc...). These functions are likely to have much in common between a ground air-traffic control system and an airborne missile firing system.

b/ Real-Time Scheduling
In a large category of process control systems, the real time scheduling of various activities relies on a small set of possibilities, e.g., run an activity at such time, or every so often, or conditional to the occurrence of a certain event. That part of the system which does the scheduling could be reused in other systems, provided it is made sufficiently

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

independent of the activities that have to be scheduled.

Reusable software components must possess certain desirable characteristics:

- simple, clear and clean interfaces: the users of a component must understand without difficulty how to use it properly;
- portability: the component is likely to be distributed on a variety of machines;
- reliability: one of the major advantages of reusing software components is that these have already been fully unit-tested;
- adaptability: a software component should be able to operate properly in a variety of similar situation; it should be insensitive to things like table sizes, data representation, etc..

3. THE ECONOMICS OF REUSABLE SOFTWARE

Reusing existing software components has a number of economic appeals. There is an obvious benefit in not developing, coding and testing something from scratch. However, this benefit must be weighed against potential problems:

- cost of acquiring the software
- cost of choosing the software
- cost of adapting to the software
- cost of adapting the software.

Other aspects must also be examined:

- impact of software reuse on traditional development methods
- maintenance problems
- distribution aspects.

There are two possible situations leading to an effective use of software components; on the one hand, there is the situation of large software developers, running several projects, and who can develop their own libraries of software component, which can grow by building up on previous project; on the other hand, there is the general (yet to be developed) market for software components, where a user can purchase a given component from a vendor. This latter situation is ultimately of primary interest:

- it can lead to improved returns on investment by allowing a developer to sell more copies of (part of) its software;
- it enables a reduction in software costs, the cost of purchasing a component being presumably less than that of making it;
- it can contribute to a better reliability of software, reused components having been tested by many more users. Some of the problems are specific to one situation or the other.

3.1. Apparent and Hidden costs

One of the most obvious costs is the purchase price if the component has to be acquired. However, the use of a foreign component in a software project may cause a variety of problems:

- instead of designing a software component, one has to choose the appropriate one. This means that the exact needs must be known, and that the choice must be made carefully. In particular, since it is often not possible to modify a component after it has been installed, an inappropriate choice may have drastic consequences.
- Another potential cost is the time spent to understand the specifications of various components before making a selection. This may also require that components have to be evaluated and tested before they are acquired.
- In most cases, the component will not provide exactly the features that the user would have wished. This does not mean that the choice was inappropriate, but it merely reflects the fact that it often costs substantially less to use something imperfect than to try and develop something perfect. Nevertheless, one of the consequences is that the system design will have to be bent to be compatible with the component. The degree to which the design must be modified is an important cost factor.
- In other cases, the system design may be an overriding constraint and one may be tempted to alter the specification of the component. First it is not always possible to do this (one may not even have the source available), and second, one must be aware of all the implications, especially in terms of testing the component again. In general, it is feasible to modify a component if it has been developed in house, so that all the documentation is available, as well as the test data.

3.2. The Distribution and Maintenance Aspects

In the case where a software component has been acquired from a vendor, a number of problems may arise when this component has been incorporated in a system which is itself distributed to users:

- can binary versions of the component be distributed freely if they are incorporated in a more complex system, or will royalties have to be paid to the authors?
- Is the component maintained? What warranty is provided? What support is available?
- In certain cases, customers of a complete system will request the sources of all software. In these cases, the source of a component must also be available, at a reasonable cost.

3.3. Adaptation Costs

"Thinking reuse" is not necessarily a natural transition, in spite of the potential advantages. In addition to an evolution of the mentalities, the development methods may have to be modified. We discuss later some of the methodological aspects of software components. It must be borne in mind that the introduction of such modifications may introduce new costs; also, the isolation of potentially reusable components may be encouraged but this may also introduce extra costs, in the form of specific component oriented reviews and requests for modifications that have no immediate use in the original project but will contribute in making the component easier to reuse.

4. DISTRIBUTION ASPECTS

In view of the various contexts for use of software components, one can envision various possibilities for the distribution of components. We consider here only the case of an independent component vendor, interested in selling software for profit (or for glory). There are two basic approaches: a conventional, protective approach, where only binary versions of the components are distributed, and a more open kind of relationship where the customer has access to the source.

4.1. Binary Distribution

Selling packages only in a binary form has the advantage of protecting the author's investment against unauthorized copies. For this reason, it also allows the vendor to quote a lower price and reach an adequate return due to a higher number of sales.

However, these considerations are essentially borrowed from the micro-computer market. Their appropriateness in the context of professional software components does not immediately follow:

- the potential market base is much narrower;
- potential customers are also professionals with a specific need: the cost of purchasing a general purpose component can be balanced against the cost of developing a much more restricted version for their own use;
- the risk of piracy in industrial organizations is probably substantially lower than in the general public;
- there are numerous technical problems in distributing binary versions of a component (described below) that make it much less likely that someone will want to make unauthorized duplications.

The technical difficulties have to do with the fact that what has to be distributed is not an object module, but something that has to be incorporated in the user compilation libraries, in particular all the intermediate tables needed by a compiler must be provided. This means:

- that the distributor must have access to all Ada compilers on the market;
- that the maintenance must be able to follow changes in the compilers;
- that not giving the source may be an impossible goal because the compiler may need an intermediate representation that can be "decompiled" to reproduce the source. One possible solution would be to use special protection modes on the tables, e.g. "execute" protection could mean that a table can be used by a compiler but not copied or decompiled. This obviously requires that the compiler itself is made to check such accesses. Also some compilers may not require the tables for the unit bodies.

The conclusion is that binary distribution is probably inappropriate for a general component distribution.

4.2. Source Distribution

Most of the problems raised previously disappear if the vendor is willing to licence the source of the components.

This approach is reassuring to the users who know that they can adapt the code if the need arises, or have the code available if mandated by a contract.

On the other hand, such a distribution is very difficult to combine with an appropriate maintenance and support if users modify the code. This is often a sufficient deterrent to prevent too many users from modifying the code.

It is also possible to have two levels of source distribution: one which is the full source code with documentation comments, etc.,

and a second level where the source that is released uses only cryptic identifiers (e.g. X00178) and no comment at all. This second level although by no means foolproof, can often be sufficient to ensure that most customers will not invest the time to "borrow" proprietary methods and algorithms from the author.

One could go even further and use a "code scrambler", i.e., a tool that transforms a well-structured program into an equivalent rather unstructured one:

Example:

```
-- This is a conventional binary search algorithm that searches
-- through a linear table for a string that matches the one given
-- in STR. The variable NEXT indicates the index of the next
-- element to be tested; it also indicates the selected item at
-- the end of the search. If the item is not found in the table,
-- the exception X_NOT_FOUND is raised.

Low  := Table'FIRST;          -- LOW and HIGH are respectively
High := Table'LAST;           -- the floor and ceiling markers
                              -- used in the algorithm.
Next := (Low + High);

loop
   case string_comp (Table(Next), Str) is

   when '<' =>
      if Next = Table'FIRST then
         raise X_NOT_FOUND;
      else
         High := Next - 1;
      end if;

   when '=' =>
      exit;        -- the item has been found!

   when '>' =>
      if Next = Table'LAST then
         raise X_NOT_FOUND;
      else
         Low := Next + 1;
      end if;

   end case;

   if Low > High then
      raise X_NOT_FOUND;
   else
      Next := (Low + High)/2;
   end if;

end loop;
```

Scrambled version:

```
        X006 := X008'FIRST;
        X007 := X008'LAST;
        X005 := (X006+X007)/2;
   <<L001>>X001 := X002(X008(X005),X009);
        if X001 /= 1 then
           goto L002;
        end if;
        if X005 /= X008'FIRST then
           goto L003;
        end if;
        raise X999;
   <<L003>>X007 := X005-1;
        goto L007;
   <<L002>>if X001 /= 2 then
           goto L005;
        end if;
        goto L004;
   <<L005>>if X005 /= X008'LAST then
           goto L006;
        end if;
        raise X999;
   <<L006>>X006 := X005+1;
   <<L007>>if X006 <= X007 then
           goto L008;
        end if;
        raise X999;
   <<L008>>X005 := (X006+X007)/2;
        goto L001;
   <<L004>>
        ...
```

4.3. Problems Related to Component Selection

One of the major problems facing a user is how to select the appropriate software component. the specifications of the component must be well understood, and in such cases the performances must be thoroughly tested before a project manager can give the go-ahead.

Giving components on loan for testing purposes is quite risky in that there is very little that can be done to control that the user has not kept an unauthorized copy if the component is not purchased.

One potential solution to this problem is the notion of a subscription service: the vendor acts as a software publisher, with a number of categories (e.g., general algorithms, civil engineering, aerospace, graphics, ...).

The customer subscribes to certain categories and is given the existing component base for the chosen categories, and keeps receiving new components during the subscription period: the central idea is that the customer does not necessarily acquire only the components that are needed now, but an entire library of components among which he can select those he wants to use; he can of course exercise them at his leisure to test the performance characteristics, etc... This allows a distribution of components on a much wider scale, and therefore lower prices per component.

5. ADA FEATURES FOR REUSABLE SOFTWARE

There are three major aspects whereby a language can support the construction and use of software component:

- the language structure must allow for the incorporation in a program of physically separate units that group together all what the component must provide, in terms of definitions, variables, subprograms, tasks, ...
- Software components must be flexible in order to be easily tailored to specific application contexts. Parameterization facilities are called for here.
- Software components are to be used in different contexts and on different machines. A language design that encourages the development of truly portable software is therefore appropriate.

5.1. Packages and Separate Compilation

The notion of Ada packages constitutes the primary mechanism for structuring programs. Packages provide for encapsulation and abstraction.

First of all, an Ada package can contain all the declarations that are related to a given concept: such declarations may have to be grouped together either for logical reasons (e.g., trigonometric funtions) or for implementation reasons, because they access the

same data (e.g., procedures to add or remove an element to a list).

The most valuable software components are those that are too complex to be worth re-implementing. As such they must offer much more than a single procedural interface. The package is therefore a convenient way to group together the various aspects of a software component. The package will be the embodiment of the component.

The separation between a package specification and a package body is a further advantage in that the user is presented with only the information required to use the component, without having to sort what is relevant from a user's point of view from what concerns only the implementation.

The language rule that makes the contents of a body inaccessible to the rest of the program forms the basis for encapsulation, i.e., preventing the user from interfering with the implementation of a component. This is necessary to enforce the integrity of the internal data structures used and the proper behavior of the component. Without this safety, it would be very difficult to provide any kind of warranty concerning an independently tested component. Note that the notion of private type is a necessary complement guaranteeing that a user can declare several objects that are administered by the package, without the possibility of compromising the internal representation of these objects.

Separate compilation is another fundamental feature: the separate compilation rules allow a user program to reference the visible declarations of a package without having to insert these declarations in its code (a reference to the name of the package is sufficient).

Example:

```
package COMPONENT is
   ...
   procedure some_action (...);
   ...
end COMPONENT;
```

```
with COMPONENT;
procedure USER_PROGRAM is
   ...
begin
   ...
   COMPONENT.some_action (...);
   ...
end USER_PROGRAM;
```

As a result, it is feasible to distribute components in binary form if necessary without forcing the user to have "external" declarations in his program, while retaining the tight checking capabilities of Ada. Encapsulation and abstraction (the fact that a component is accessible only through its "logical" interface, without reference to any implementation intricacy) have also an important impact on maintenance: new versions of a component can be released without requiring any modification to the user programs, or even any recompilation --only relinking is necessary, and even that could be avoided on certain systems, e.g., Multics. Also components can be available in multiple versions corresponding to a common specification, leaving the user free to switch from one version to another if he sees it fit.

5.2. Generics

In some cases, a component that has been developed can be reused as is in another context. However, in most situations where reuse is contemplated some adaptation is necessary. This will be in particular the case whenever a component is supposed to operate on certain user defined types. A typical example would be a set of functions that operate on fixed point data: in Ada, there is no single fixed point type; rather, the user must declare a type based on his own range and precision requirements, e.g.,

```
type MEASURE is delta 0.0005 range -273.00 .. 10_000.0;
```

From such a declaration, the compiler will determine the appropriate representation. Now suppose that an interesting component has been written, that performs some complex computation on a fixed-point number, e.g. taking the logarithm. The component itself could not have been written for this specific type, which was not even known when it was developed. So, what is needed is a way to parameterize the component in terms of a user type. This is exactly what is provided by the generic facility: by writing

```
generic
   type ANY_FIX is delta <>;
function FIX_LOG (X: ANY_FIX) return FLOAT;
```

the writer of the function will permit its use with any user-defined fixed-point type. Parameterization facilities also allow the use of user-defined variables and subprograms inside a software component.

A large class of components can make direct use of these facilities in particular components that define and manipulate data structures (lists, queues, stacks, trees, graphs, ...) or general purpose algorithms (e.g., sorting and searching).

In essence, a generic unit is a template for what can become a real unit when all the generic parameters are determined. In the above example, FIX_LOG is not a function that can be called directly by the user. Rather, the user must first create an actual function from the template; this is called *instantiating* the generic unit:

```
function MEASURE_LOG is new FIX_LOG (MEASURE);
```

After the instantiation, MEASURE_LOG is treated exactly as a function that would have been declared locally.

Both packages and subprograms can be turned into generic units.

This feature makes an existing component directly usable in many more contexts than it would have been without the parameterization. This is obviously an important factor in determining the economic benefit of writing a software component.

5.3. Portability Features

There are three major elements that can result in a poor portability of software:

- differences in the representation of types;
- use of non-standard features, or system-dependent libraries;
- use of "dangerous" programming practices, such as violating the types of values (a "violation" of a type occurs when the bits forming a value (e.g., an integer value) are interpreted as if they represented a different value (e.g. a floating point value).

Non-portability could be a severe hinderance to the large dissemination of software components. Fortunately, several of the above issues have been addressed in Ada.

5.3.1. Control over Type Representation

A programmer can force a certain representation of the values of a type through representation clauses. Also, and equally important, the numeric types, that are often a big source of non-portability are expressed in Ada solely with respect to the desired properties of the values, in terms of range and precision, independently of an actual physical implementation. Thus a user can declare:

```
type MY_INT is range 0 .. 200;
type MY_FIX is delta 0.0002 range 0 .. 86_400.0;
type MY_FLOAT is digits 9;
```

The given information is sufficient to let the compiler select a representation that is sufficient to accomodate the required properties; In this way, the programmer is relieved from the headaches caused by different word lengths on different machines.

5.3.2. Use of non-standard Features

Because the Ada language itself is standardized, there is no real non-standard feature. On the other hand, there may be a variety of specific components or libraries available on a given system, upon which a user program relies. Little can be done to prevent that. However, one can hope that widely used components written entirely in Ada become generally available, pretty much in the same way as the portable "C" library.

5.3.3. Use of Dangerous Practices

Type violations, randomly accessing the memory, and similar interesting practices are extremely restricted in Ada. They can be done, but not without the user explicitly saying so.

For instance, to use an integer as a floating point value, one would have to reference and instantiate the predefined generic function UNCHECKED_CONVERSION, e.g.,

```
function INT_TO_FLOAT is new UNCHECKED_CONVERSION (INTEGER, FLOAT);
```

and then use it explicitly, e.g.,

```
INT_TO_FLOAT (X);
```

Accessing a specific address in memory would require an address clause, such as:

```
for X use at 16#FFF0A#;
```

These low-level aspects both restrict the use of dangerous programming practices, and flag them for attention when porting is attempted.

Through its package structure, its separate compilation facilities, the flexibility offered by generic units and the possibility of writing truly portable software, Ada will permit the generalized practice of reusing components in software systems, and will also be a determining factor in the establishment of a real software component industry.

6. PROGRAM CONSTRUCTION AND SOFTWARE COMPONENTS

The possible gains offered by an effective reuse of software components may lead to a modification of the way software systems are built. One must consider the potential for using existing components when designing a system; it is also advisable to scrutinize newly developed software for possible "componentization", i.e., slightly modifying the specifications of what has been designed so that certain parts can become reusable components.

6.1. Using Existing components

The simple attitude towards reusing software consists in specifying and designing a system, and looking for possible components to fill in the slots traced by the designer. In such a case, the potential for finding exactly the appropriate component is rather low, and may be time-consuming. The desired component might not exist, or may be found in some especially exotic place (like Sophia-Antipolis, for instance), where people would not search for it.

With such an attitude, only two kinds of components may be reused: those that are already well-known in-house, and those that are fairly big and correspond to a standard (e.g. a graphical or data base interface).

Many more parts of a system could probably be implemented by using existing components if the designer can make his choices based on the availability of these components.

It must be kept in mind that component selection must be done early in the design, because the specifications of a given component may have an influence on the architecture of the system in which it is embedded.

In order to do this, a special activity must be launched at the start of a project, which consists in establishing a catalogue of relevant components for the project, possibly acquiring the most promising ones to evaluate and test them. This could be done using a list of keywords that relate to the project, deriving them from the requirements analysis. Such an activity is a fairly specialized one, and one could expect to see a "component engineer" as a particular profile in a project team.

The work of finding the relevant components can of course lead to specialized tools, i.e., component data bases, that can be interrogated by keywords, performance information, protocols, etc...

6.2. Componentizing Software

An important part of the component base that will be effectively reused is likely to be internal to a company; such a base is built gradually by accumulating components that had been originally designed for a specific system.

However, if a system is constructed without special attention, a variety of things may hinder the extraction of true components: the interfaces may be poorly packaged, the component may be relying on some global data that is at the heart of the system, or it

may make assumptions on how it is going to be used which may not be valid in a different context. The design is also likely to be too specific, e.g., operating on a specific data type where this could be parameterized through use of the generic facility.

If there is a strong will to develop a component-based approach, part of the review process that goes on in a project must be devoted to identifying those parts that could lead to reusable components: during the global design, modifications can be made to the interfaces, to promote encapsulation, to group together related objects or, to the opposite, to separate things that are not related, and to eliminate unnecessary dependencies. During detailed design, one should also correct certain options that could hinder the extraction of a component. Lastly the potential components that have been identified must be given to separate teams for realization and testing, as it is important that they become an independently maintained product very quickly. Here again, the quality assurance team could incorporate a specialized "component adviser", whose specific objective is to look for potential components in a software project.

7. CONCLUSION

Ada offers a unique opportunity to reach a significant productivity improvement in the software through an effective reuse of software components. Specific features of Ada are central to such an evolution: the package structure, separate compilation rules, the generic facility, and the improved portability of Ada software.

One should expect that such an evolution will be gradual, as mentalities will have to adapt to the full potential offered by the reutilization of software components. Bases of software components will have to be built and made accessible on a commercial basis, leading to an industry of software components.

Such an industry will have to face fairly high investment costs, and be aware of the specificities required for the effective distribution of software components. Once the appropriate structures are in place, both inside companies and on the open market, components may have a profound impact on the way software is developed.

ADA COMPILER DEVELOPMENT

Dan Eilers

President
Irvine Compiler Corporation
18021 Sky Park Circle, Suite L
Irvine, California, 92714

Summary

The ICC-Ada compiler has enjoyed considerable success in flight applications. Much of this success is due to careful consideration during its development of the issues which are critical to embedded systems applications. Compiler development by nature involves various trade-offs, Ada compilers in particular. Understanding these issues and trade-offs is important to the selection and use of an Ada compiler for embedded system applications.

1. Introduction

Ada was not designed to be easy to compile. If so, there would be no reason for this paper. Ada compilers were intended to be written only once, and used forever, so the focus in language design was to make life easy for the programmers using Ada. This is in sharp contrast with the design of most other languages, including Pascal, on which Ada was loosely based. In addition to the many difficult language features and their interactions for a compiler developer to address, many design decisions must be made, affecting the usability of the compiler for various purposes. It is therefore important for the user to be aware of the goals of the compiler developers, and the tradeoffs they have made in developing their compilers. Some of the major issues facing a developer are portability, compile-time resource requirements, optimizations, run-time environments, interfaces to various other tools, and customer support. Each of these will be addressed in turn.

No single compiler can possibly do everything right for every user. But then not all users need use the same compiler. Fortunately a number of different compilers have been developed, and with widely diverging goals. Some excel at optimization, some at compile speed, some at error and warning diagnostics, some at library management, some at environment tools, etc. It may even be appropriate for a single user to use one compiler for training, another for rapid prototyping, another for PDL processing, another for full scale development, and then hand the project off to be maintained on yet another. They may even all run on different computers. The strict standardization of Ada makes this feasible.

Despite this degree of standardization, compilers used for generating production code are not particularly interchangeable. Each has its own unique run-time systems, and implementation-defined pragmas and attributes for dealing with embedded applications. And each is designed to work with particular assemblers, linkers, debuggers, floating point processors, timers, i/o devices, memory models, etc. Since various implementations do different optimizations, coding rules for creating efficient code will vary accordingly. In addition, the language-defined low-level mechanisms are not well tested by the validation procedures, and have in fact been omitted by many implementations. Selecting a production compiler therefore deserves careful consideration.

2. Portability

For some compiler implementers, the decision of which host and target environments to support is a non-issue. Mini-computer manufacturers for example, will host their compilers on their own systems. But even they typically sell a range of equipment, and usually more than one host operating system. Independent compiler vendors typically try to support as many different host/target combinations as possible. This has benefits both for the implementer as well as for the user. The implementer benefits by amortizing the compiler development costs over a broad user base, and the user benefits by not getting locked into a particular brand of hardware for software development. In some instances it is possible, and very convenient to target to several different architectures from the same development host.

Designing a compiler to be rehostable entails several constraints, none of which are severe. It means not relying on "compiler-compiler" tools which are only available on particular systems. It also means not relying on particular text file formats or file-system operations. It also rules out reliance on process mechanisms for such things as implementing the library management system. Tricky, non-portable coding techniques cannot even be considered. And an implementation language must be chosen for

which compilers are widely available.

The choice of an implementation language is critical. It usually boils down to using the compiler to compile itself, or using some other language. The implementers who have chosen mature languages have so far ended up with the most efficient implementations. It remains to be seen whether the others can do as well using Ada. ICC has taken a middle ground approach. Our compiler is written in an extended Pascal dialect, but uses it's own code generator, by sharing the internal representation with the Pascal compiler. This results in an efficient implementation, avoiding any Ada-related inefficiencies, but also one that is easy to rehost. Rehosting is done simply by retargetting the code generator, and then recompiling. No lines of source code need be changed.

Retargetability of compilers in general is a fairly well understood technology. A low-level intermediate language is used which can be efficiently mapped onto various target instruction sets. DIANA, by contrast is a much higher level intermediate language for Ada, used primarily as the interface between the front and middle "ends" of some Ada compilers. Retargetability is a poor excuse for slow compilers, since they all generally use similar intermediate representations. Retargetable compilers have an added reliability advantage in that some subtle bugs typically show up (and are fixed) each time the compiler is retargetted.

3. Compile-time Resource Requirements

Probably the major resource issue, because it directly impacts rehostability, is memory utilization. It may be possible to build a 4K or 8K BASIC (although most current versions are much larger), but not an Ada compiler. The best to have been done so far was an implementation for the 128K byte Western Digital Micro-engine, although this compiler had severe restrictions on program size. Potential host machines can be divided roughly into three categories. First is 8-bit micro-computers and older 16-bit mini-computers with addressable spaces of 128K bytes or less. Although millions of these machines exist, and it would be desirable to accommodate them, very little can be done. Not only is their main memory too small, but their disc space, and processing throughput are usually impractical for Ada as well. It is however feasible to cross-target Ada compilers to these computers, although care must be taken not to fill their entire address spaces with the run-time system.

Some people used to think that the small host machine problem could be solved by subsetting the language. However, not only was this severely frowned upon by the AJPO, but it turned out to be impractical. Ada is too tightly integrated to be subsetted cleanly. For example, the text_io package depends on almost every feature of the language except tasking and subunits, and even these could be used. Text_io uses overloading, generics, exceptions, derived types, default parameters, unconstrained arrays, renaming, packages, etc. And not only does text_io use the full language, but even beginning users do the same, for example unwittingly using derived types in a simple integer type declaration. Tasking and subunits could be cleanly subsetted away, but with only modest savings in compiler size. Most other cuts would leave a different language, not a true subset.

Even though Ada can not be cleanly subsetted, it could however be simplified, although not to the extent of making compilers significantly smaller. The language contains hundreds, maybe thousands of restrictions, many of which serve no useful purpose. At best they clutter compilers. Worse, they frustrate users. Some, such as the restriction that functions may not have 'out' parameters, are vestiges of former versions of the language that in this case attempted to prevent functions from having side effects. Other restrictions were intended to make implementations easier, or exist because no use could be thought of for a construct. But many of these have been shown to be misguided. And yet others, such as the restriction that case expressions must be discrete, apparently exist simply because it has always been done this way.

Other simplifications that could be done involve eliminating syntactic ambiguities and inconsistencies. For example, the string literal "+" as an actual parameter in a generic instantiation could really be a string literal, or it could be the name of the addition operator, depending on context. An example of an inconsistency is that a task's priority is set by a pragma, its storage size is set by a length clause, and an object's address is set by an address clause, all with different syntax. Compiler developers are rarely content to leave the language design to others...but I digress.

At the opposite end of the computing scale are the powerful 32-bit (or more) machines with virtual memory, or at least several megabytes of addressable space. Supporting Ada on these hosts is not a problem as far as memory is concerned, although surprisingly enough, due to contract requirements, some compilers designed only for this class of hosts have elaborate mechanisms for minimizing main memory usage. On these hosts, memory management is best left to the operating system.

The final category is machines with approximately 512K-1M bytes of main memory. This includes the current generation of 16-bit micro-computers as well as some older large scale computers with 18-bit address spaces. These constitute the interesting case, both from a technical standpoint, and from the perspective that software development may be migrating from typically mini-computer development systems, to networks of distributed work-stations. Ada compilers can be implemented for these

machines, if memory usage is minimized.

Memory usage for a compiler consists of three components: code space for the compiler itself, data space for the symbol table, and data space for the parse tree. Code space can be minimized in three ways: by using an interpreter to take advantage of compact stack oriented code; by segmenting the compiler and using overlays; or by splitting the compiler into multiple independent passes. Interpretation is feasible, and has in fact been used for some implementations, but slows compilation considerably. The use of segmentation is not particularly feasible, since compilers tend to frequently hop from one area to another. For this reason, it is important on virtual memory systems to have sufficient real memory to minimize thrashing with any compiler.

Using multiple passes is the most common approach, but it too has its drawbacks. Foremost of which is the i/o needed to reconstruct both the symbol table and the parse tree. This i/o time is usually the bottle-neck limiting compilation speed. And not only does it take time, but it also adds a significant amount of code to the compiler to read and write these structures which have many different kinds of nodes, which defeats the entire purpose. These structures could potentially be kept in memory between passes, but that also defeats the purpose of minimizing memory usage. The conclusion is that not much can be done practically to minimize code space requirements, with the possible exception of using a good compiler to compile the Ada compiler.

Some implementations attempt to control symbol table space usage. On the surface this is very attractive, since symbol table space requirements can be quite high in Ada. Packages such as text_io for example have large interface sections which are made visible by a 'with' clause. Multiple instantiations of generic packages can also rapidly use up space. And seemingly simple derived type declarations often implicitly declare a number of functions to operate on objects of the type. However there are major drawbacks here too. Overloading rules for example require the compiler to search the entire symbol table for alternate interpretations of subprogram calls, even for such seemingly simple expressions as "2+2". Having to page the entire symbol table in and out of memory just to interpret such common expressions could be prohibitively expensive. Also the necessary checking to determine whether a particular symbol definition was in memory every time it was referenced is costly both in time and code space.

Another way to reduce symbol space requirements is to use ordinals instead of pointers to reference symbols. Where pointers are likely to take up a full 32-bit word, ordinals can be 16 bits, or possibly sometimes less. This approach has two drawbacks: one is that table sizes place arbitrary limits on the number of symbols of particular classes in particular scopes; the second is that references to the symbols necessarily are likely to be less efficient in both time and space than having a pointer directly to the symbol. Since Ada was designed for large programs, any arbitrary symbol table limit is almost guaranteed to be exceeded at some point.

ICC makes neither of the these attempts at minimizing symbol space usage. The penalties in terms of compilation speed or arbitrary limits on number of symbols were felt to be too great. Instead ICC relies on minimizing code size, and eliminating parse tree data space requirements in order to run the compiler effectively on work-stations. Data space for parse trees is eliminated by parsing one statement at a time, generating the appropriate nodes to the intermediate file, and reclaiming any storage used in the process. This allows arbitrarily long files to be compiled, which is particularly useful when large FORTRAN modules, for example, are translated to Ada without being broken into separate modules.

Ada was not really designed for this kind of one-pass, statement at a time, mode of compilation. The lack of a requirement for declaring labels, for example, has the unfortunate result that a statement can make a previous statement illegal. Also, in-memory parse tree representations are very handy for many of the required semantic checks, and for generic instantiations, but all these problems have solutions.

4. Optimizations

Optimization of Ada compilers is likely to take longer than anyone expects. There is much to be done. And the designers of Ada practically assumed the existence of significant optimizations. One of the major differences between Ada and typical systems programming languages, is that Ada emphasizes portability, and the others emphasize the ability to directly manipulate the hardware at the lowest possible levels. In order for Ada to achieve similar results, a large number of optimizations must be performed. And with Ada, unfortunately, it is not always easy to tell for a given compiler which constructs generate efficient code, or under which circumstances. Slight changes to a program can have significant effects, such as when a static constant is made dynamic.

Optimizations fall into two broad categories. Those that are to be performed in the front-end of the compiler and those which are code generation issues. Code generation issues are fairly well understood. They are generally independent of the input source language. This means that code generators and optimizers developed for other compilers for other languages can readily be adapted to Ada compilers. Although they still need significantly more work tailoring them to recognize the common inefficiencies created by Ada front-ends.

The more difficult category consists of generating reasonable intermediate code for each of Ada's constructs, since there is only so much an optimizer can do with what it's given. Each of Ada's features requires special attention. Often there are several ways of implementing the feature, each better in certain cases. Some implementations will support only one. Others will allow the user to control the implementation via a pragma, and in other cases, the compiler will attempt to determine the best implementation.

In exception handling, for example, the overhead can be placed either at the time of a raise, or at the time any scope is entered or exited, or possibly at the time a scope is entered only if it contains a handler, or some combination of these. No single method is best for all applications. In generics there are obvious tradeoffs between space and time, depending on whether or not generic bodies are shared, and under what circumstances. Discriminant records containing dynamic arrays can be implemented in a variety of ways, depending on whether it is more important to efficiently access fields of the record, or to copy and compare the records. Descriptors for unconstrained arrays and discriminant records can be created with the data they describe, or only on demand, or can be shared when possible. Bit operations are not directly available in Ada, and can only be effectively used if logical operations on packed arrays of bits generate single machine instructions.

Tasking will require numerous optimizations, depending on what the purpose of the task is, or on how many tasks will be running at a time, etc. Simple tasks used only to synchronize accesses to shared objects need not involve the full complexity of the scheduling algorithm. And real-time applications such as flight control which may simply switch back and forth between a foreground task and a low priority background task can also theoretically involve almost no task-switching overhead. ICC has much work remaining in most of these areas.

5. Run-time Environments

Run-time environments are an area of great importance for embedded applications, and Ada compiler run-time systems can vary greatly. Of prime importance often is the size of the run-time system. A related factor is its decomposability, so that no space is taken by something which is never used. Other concerns are support for handling interrupts, time-slicing and task scheduling, asynchronous or non-blocking i/o, various forms of automatic garbage-collection, and the ability to run more than one Ada program simultaneously on the same processor, with inter-process communication. Tailorability of the run-time system is also very important, particularly in unique embedded systems, whether it "voids the warranty" or not.

Another run-time issue becoming more important is the ability to run Ada code on unique architectures, such as multiple-processor environments, either tightly or loosely coupled. Support for accessories such as array processors, or other such "peripherals" is also important to some classes of users. Again ICC does not claim to have the the final answers, although having virtually its entire run-time system written in Ada makes customization easier.

6. Tools

An Ada compiler is but one member in a family of tools, although a very important member. Actually, possibly too much effort was expended early on in the Ada program on environment issues, and not enough on the compilers themselves. Compilers were thought to be well understood technology, requiring little attention. They were also mistakenly thought not to fit well in existing environments. In the early stages of Ada's application to embedded systems, a good compiler with a functional environment is likely to be more useful than a good environment with a functional compiler.

However, as the compilers mature, the environment tools will increasingly gain in importance. Also, the larger the project, the more important the tools become. And Ada has a way of vastly complicating the tools required. For example, typical debuggers were not designed to support Ada tasking. Overloading causes problems uniquely referring to subprograms. And arbitrarily long identifier names with full significance are unheard of. An ironic aspect of complicated Ada tools, particularly the compilers themselves, is that they can least afford to incur the additional overhead costs of layered kernels, such as CAIS, which are designed specifically to aid these tools.

The library management system for a large Ada project is probably the most critical tool. It must allow for such things as hierarchies of program libraries, with package specifications under configuration control, and package bodies which can be modified in a local context, debugged, and integrated with others. Minimizing the number of recompilations required by any given change is also important on large projects, but not at the cost of compilation speed. Ada's requirements of full checking across compilation boundaries create enormous burdens on a library system, that earlier languages knew nothing about. Other complications involve libraries scattered across multiple host computers, and libraries that support multiple versions of a compiler, both for the same target or for different targets.

7. Support

An unfortunate side effect of Ada's complexity is that even the sophisticated users will generally prefer not to do their own compiler support. With other languages, critical bugs could be fixed immediately, and optimizations added. But in an Ada compiler, fixing one bug is likely to cause several new ones. Even most mini-computer vendors, who would otherwise write their own compilers, are instead adapting portable implementations. And Ada compilers will always have bugs. Users still may find it feasible to make code-generator modifications to their Ada compilers, but should be alert to the issue of vendor support.

8. Conclusion

The design and implementation of an Ada compiler is a very demanding project, and likely to be never ending. Along the way, many design decisions are made affecting the usability of the compiler for various applications. Despite validation, all Ada compilers are not created equal, nor is it necessarily easy to switch from one to another for embedded applications, and they are not free of bugs. Careful compiler selection is called for to reap the benefits that have been painstakingly designed into the language, its compilers, and programming environments.

ADA[1] COMPILER VALIDATION

by

Erhard Ploedereder, Ph.D.

Tartan Laboratories

477 Melwood Ave.

Pittsburgh, PA 15221

U.S.A.

Summary

This paper discusses Ada Compiler Validation from the viewpoint of both the compiler supplier and the compiler user. The objectives of the requirement for validation and the resulting benefits, as well as the limitations of validation are presented. The process of Ada compiler validation is detailed along with its various problems as well as issues relating to validation and the use of validated compilers. Solutions to these problems and issues are explained, as they are proposed in a recent AJPO draft of a revised validation policy.

Preface

The proliferation of languages and language dialects has been recognized as a major contributing factor to the high cost of software development and maintenance. Problems that arise out of such proliferation include the added expense of maintaining a software environment for each such language and dialect, the cost of educating personnel in the use of many different languages, and the increased cost of porting software to other systems.

The programming language Ada was developed by the US Department of Defense (DoD) to become the only programming language to be used in DoD mission critical computer resource (MCCR) projects. Ada has been standardized and mechanisms have been put in place to prevent the existence of language dialects. The validation of Ada compilers is an integral part of these mechanisms.

This paper discusses various aspects of the Ada compiler validation, including its objectives and limitations, the actual process of validation, the policy for the use of validated Ada compilers within U.S. DoD, and the economics of compiler validation.

1. Objectives of Validation

Past experience with standardized languages (e.g., Jovial, Pascal, Cobol), has shown that the mere existence of a standard is insufficient to prevent the emergence of language dialects, as compiler vendors tend to deviate in their compiler implementations from the respective language standard. As a consequence, software developed with one compiler often cannot be compiled with a compiler supplied by another vendor without major changes to the software. As a result, porting of software between systems employing different compilers becomes expensive, if not economically infeasible. Particularly in the military setting, large software systems have a lifespan of many years, during which enhancements and new hardware developments will make it necessary to port the software to new underlying systems.

Recognizing this problem, the U.S. DoD has taken steps to insure that the mistakes of the past are not repeated in the Ada effort. It has obtained a trademark on Ada and requires that the conformance of compilers to the language standard, ANSI/MIL-STD 1815a, be ascertained by means of the validation process, before such compilers are allowed to be called Ada compilers. The objective of this requirement is to prevent language dialects and thereby to enhance the portability of software written in Ada.

[1]Ada is a registered trademark of the United States Government, Ada Joint Program Office

2. *The Meaning of Validation*

Validation is **not** a replacement for acceptance testing by the buyer of an Ada compiler. Staying true to its objectives, the validation process tests only for conformance with the language standard. It does not address issues of usability or efficiency of the compiler and the generated code. For example, the first Ada translator ever validated was the New York University Ada Interpreter, which was a remarkable achievement as an experimental system for semantic modelling and for teaching Ada, but definitely not a system to support application code written in Ada.

Moreover, validation **is not** a guarantee of compiler correctness. Because the validation process employs testing by examples as the test method, only part of the intricacies of the Ada language can be covered by the validation.

On the other hand, the test suite of the validation process is sufficiently large and well designed that broad coverage is achieved. Successful processing of the test suite is a significant and major milestone for compiler developers on the way to compiler correctness and conformance to the language standard. As such, it significantly raises the users' confidence in the correctness and completeness of Ada compilers.

Notwithstanding validation, users must evaluate the appropriateness of a validated compiler for their particular needs along all dimensions, such as code efficiency, usability and user friendliness, interaction with the environment, provision of features that are optional in the language standard, etc.

3. *The Validation Policy, Procedures, and Guidelines*

In January 1986, AJPO drafted a new validation policy, consisting of three documents:

; the general policy for compiler validation;

; the procedures for the conduct of the Ada validation process; and

; the policies and guidelines for the use of Ada compilers in DoD.

The first document establishes the policies for the validation process applicable to Ada compilers both for general trade and for U.S. DoD applications. The second document details the procedures necessary to establish a compiler as a validated compiler. The third document integrates the requirement for validation of compilers with the constraints of life-cycle management procedures in U.S. DoD projects. While this third document relates specifically to the use of Ada in U.S. DoD, similar issues will arise in general trade usage of Ada compilers. It therefore can act as a model for general trade companies in adopting similar policies and guidelines.

In the subsequent sections, the provisions of the draft policy are described in more detail. Since, at this time, the draft is still under discussion and the policies and procedures not officially issued, changes may occur that are not reflected in this paper and may contradict its contents.

4. *The Components of the Validation Process*

The three organizational components of the validation process are the Ada Joint Program Office (AJPO), the Ada Validation Organization (AVO), and the Ada Validation Facilities (AVF). The technical component is the Ada Compiler Validation Capability (ACVC). These components and their role are briefly described in the following sections.

4.1. The Ada Joint Program Office (AJPO)

The Director of the AJPO formulates the validation policy, issues charters to AVF, and is ultimately responsible for all components of the validation process. Validation certificates are issued or authorized by AJPO after successful completion of the validation process.

4.2. The Ada Validation Organization (AVO)

AJPO has chartered an Ada Validation Organization (AVO) with the task to establish detailed guidelines and procedures for the process of Ada compiler validations, and to supervise Ada Validation Facilities (AVF) in their conduct of the validations. The AVO consults to AJPO on validation issues that arise from compiler validations. Furthermore, the AVO maintains the lists of validated compilers.

4.3. The Ada Validation Facilities (AVF)

The AVF are impartial organizations that are chartered by AJPO to conduct validations. Such charters are in effect for three years and are renewable. AVF are bound to adhere to the validation policies issued by AJPO and to the procedural guidelines established by the AVO. The AVF act as the interface between suppliers of compilers that desire validation of their compilers and the certification body comprising AJPO, AVO, and AVF.

Currently, five AVF are in existence: two in the U.S., and one each in Germany, Britain, and France. Their addresses are listed in Appendix B.

4.4. Ada Compiler Validation Capability (ACVC)

The AJPO has procured the Ada Compiler Validation Capability (ACVC) as the technical means to ascertain conformance of Ada compilers to the language standard. The core element of the ACVC is a large set of test programs to be submitted to the compiler to be validated. Currently, this test suite consists of about 2400 programs, each of which checks particular aspects of the language. About half of the tests are programs that contain errors that a compiler must detect. The remainder of the tests are correct Ada and must be executable after compilation. Some of the tests examine the correct implementation of optional features of the language, which a compiler may or may not support; the applicability of these tests depends on the individual compiler. The tests containing errors usually contain a large number of variations of the same class of error. Each of these errors must be recognized by the compiler. Since error recovery occasionally will mask subsequent errors, such tests may have to be split into multiple tests to isolate individual error situations, thus further increasing the number of test examples used in the validation process. The result of the tests containing errors is obtained by examining the listings produced by the compiler. Executable tests, on the other hand, are carefully designed to condense an indication of success or failure into a small print-out produced by the execution of the test.

The ACVC is augmented by software tools that facilitate the evaluation of test results, and by the Implementers' Guide which is a detailed commentary on ramifications of the Ada language semantics and on the corresponding ACVC tests.

At any given time, three versions of the ACVC test suite are in existence:

1. the development version: this version is not released to the public. New tests and corrected tests are added to this version.

2. the field-test version: this version is available to the public. During the first three months of its release, individual tests may be corrected. After three months, this version is frozen and can be optionally selected by a compiler implementor to be used in the formal validation process.

3. the released version: this frozen version of the test suite is generally used during formal validation. Tests in the frozen versions that are found to be incorrect are withdrawn and, after being corrected, are added to the development version.

The field-test and released versions are available to the public. Periodically the released version is retired and replaced by the field-test version; the development version becomes the field-test version and a new development version is created. Currently, this change of versions occurs in six-month intervals on June 10th and December 10th of each year. Consequently, for formal validation, implementors have a choice between the field-test version and the released version from September 10th to December 10th, and from March 10th to June 10th. During the remainder of the year only the current released version of the ACVC test suite can be used in formal validation. The overlap of ACVC versions admissible in validations has been introduced to permit compiler suppliers to reduce their risk of failing to complete the validation process before the released version expires; by choosing the frozen field-test version for validation, potential delays

can be absorbed more easily.

As the ACVC is rapidly maturing, it is expected that, in the near future, the release cycle will be lengthened so that the released version of the ACVC test suite has a lifetime of nine or twelve months.

5. *The Validation Process*

Ada compiler validation requires that all applicable tests of the ACVC test suite must be passed successfully by the compiler to be validated. Thus, validation testing is a pass/fail decision. The evaluation of the test results is done by an impartial body, i.e., an AVF, in two major steps: first, the AVF examines the test results as they were submitted to the AVF by the compiler supplier; at this stage, testing issues that the supplier or the AVF might raise, such as the applicability of individual tests, are resolved. Second, the AVF witnesses the on-site testing of the compiler, the results of which formally determine the success of the validation. Normally, any causes for potential failure of the validation testing are discovered and corrected during the first step so that on-site testing becomes a mere formality.

The applicability and correctness of individual ACVC tests may be challenged by the compiler supplier. Applicability issues arise for tests that examine optional features of the language not supported by the compiler. Correctness issues relate in most cases to tests for which the test designers interpreted the language standard differently than the compiler writers. These challenges are forwarded by the AVF to the AVO which, with the help of a group of language experts, determines whether the compiler writers' interpretation is permissible by the language standard. If so, the test is withdrawn from the test suite. The compiler supplier is informed about the disposition on the challenge typically within two weeks.

After successful validation, a Validation Certificate is issued by AJPO to the supplier of the compiler. This certificate is valid for one year and entitles the supplier of the compiler to market the compiler as a validated Ada compiler. The Validation Certificate attests to the successful completion of validation testing for a specific compiler and specified host and target systems, for which the full ACVC test suite was run as part of the on-site testing. The validated compiler for the specified host and target system is referred to as a "Base Compiler".

In detail, the validation process consists of ten steps that must be successfully completed in order to obtain a Validation Certificate for a Base Compiler. These ten steps are explained in Appendix A.

6. *Revalidation*

In order to preserve the status of a Base Compiler as a validated Ada compiler after expiration of its Validation Certificate, the supplier must re-submit the compiler for validation prior to the expiration of the Validation Certificate.

Such revalidations have a dual purpose: first, they guarantee that changes to compilers do not cause a gradual deviation from the language standard. Second, the ACVC test suite is continuously enhanced to provide more comprehensive coverage of the language standard. The requirement of revalidation insures that compilers conform to the standard even in those areas that were not covered by the ACVC version that was in effect at the time the compiler was initially validated.

It is expected that, as compilers and the ACVC gradually mature, the validity of the Validation Certificate will be extended to two years, thus reducing the effort and cost of revalidations.

7. *Status of Maintained Compilers*

Ada compilers will undergo maintenance changes and enhancements, in particular during the initial period after their first release. As it is economically and administratively infeasible to require formal revalidation after each change to a compiler, as well as undesirable to discourage maintenance upgrades by such a requirement, AJPO has adopted the policy that the validation status of a compiler automatically extends to its revisions. This policy relies on the mandate for annual revalidation to detect deviations from the standard and on the fact that it is in the best interest of the compiler suppliers to remain conforming to the language standard.

8. *Compilers for Multiple Configurations*

A particular problem with which compiler validation is confronted is the issue of compilers for multiple host and target system configurations. Typical examples are compilers hosted on and targeted for system architectures that are compatible in their instruction set; for such compilers, it can be reasonably expected that a compiler tested on one such configuration will generally execute correctly for any other configuration within the family of these architectures. This expectation is more difficult to justify if there are minor differences in the instruction set architecture or the operating system. A similar complication arises from compiler switches that influence execution characteristics of the compiler or the code generated by the compiler.

Clearly, it is quite impossible to perform compiler validations for all host and target configurations for which a compiler is designed, and for all switch settings of the compiler, due to the combinatorial explosion of all permutations of these influencing parameters. On the other hand, it is equally impossible to decide with certainty whether these parameters will alter the outcome of validation testing without actually performing the test. In a strict sense, it is therefore impossible for the AVF to attest to the successful completion of the validation testing, unless the tests are executed on the specific configuration.

The draft validation policy addresses this issue by introducing the concept of a "Registered Derived Compiler": A compiler supplier who has successfully validated a Base Compiler can register with AJPO additional compilers derived from this Base Compiler. Thus registered compilers are considered as equally validated compilers. This status is tied to the validation status of the Base Compiler and expires with the expiration date of the Validation Certificate for the Base Compiler. A registered derived compiler can be marketed as a validated Ada compiler.

As part of the registration process, the supplier has to assert that the compiler is a - possibly modified - version of the Base Compiler and that the compiler conforms to the Ada language standard. AJPO or AVO may require substantiating information for this assertion, such as a hardware vendor's statement guaranteeing the equivalence of the instruction set architecture or the results of running ACVC tests. However, final judgement on trusting a Registered Compiler to pass all applicable ACVC tests lies with the prospective buyer of the compiler.

The AVO maintains the list of Registered Compilers. If, at some later time, it is established that a Registered Compiler does not pass an ACVC test that the Base Compiler passed successfully, the compiler will loose its status as a Registered Compiler, unless the supplier corrects the problem. Any such challenges, which may be brought to the attention of the AJPO by any current or prospective user of the compiler, will be noted in the list of Registered Compilers after ascertaining the validity of those challenges.

The intention of this policy is to avoid the combinatorial explosion of formal validations, without requiring AVF or AVO to render judgement on whether a compiler is likely to pass validation testing on some host and target configuration without running the test suite on this configuration. It relies on the market place to force Registered Compilers to adhere to the language standard as well. Given that registration of compilers is dependent on the existence of an already validated Base Compiler by the same supplier, it is reasonable to assume that any potential deviations of Registered Compilers from the language standard are minor, unintentional, and easy to correct.

9. *Life-Cycle Management of Validated Compilers*

The mandate for the use of validated compilers in projects is generally in conflict with the practice of baselining compilers for extended periods of time. Since the validation status of a compiler expires after one year, annual revalidation is necessary. Due to changes in the ACVC test suite, it may be the case that the baselined compiler fails revalidation; an upgrade of the baseline would be required. For project stability or contractual reasons, such upgrades may be highly undesirable.

A revised policy for the use of Ada compilers in DoD projects has therefore been drafted that reconciles the requirement for compiler validation with the need for baselining compilers. This policy introduces the concept of a "Project-Validated Compiler".

A Project-Validated Compiler is a validated compiler that has been baselined in accordance with DoD life-cycle management policies. Such a compiler maintains its status as a Project-Validated Compiler throughout

the duration of the project. Only major system upgrades require that the baselined compilers also be upgraded to Validated Compilers. The rationale of this project-specific extension of the validation status is that, as the ACVC is maturing rapidly, it is very unlikely that a validated compiler would exhibit grave deviations from the language standard. The benefit of preventing minor accidental deviations by enhancements to the ACVC is low compared to the complications that arise from the risk of forcing an upgrade of the baselined compiler at inopportune stages of a project. Naturally, program managers are encouraged to plan for periodic upgrades of the baseline to a compiler with a Validation Certificate in effect.

While the Ada compiler used during development of MCCR software is not required to be a Project-Validated compiler, the compiler used for the software delivered for operational testing must be a Project-Validated Compiler. The acceptance testing upon delivery of such software will ascertain that the compiler passes all applicable tests of an ACVC version equal to or, optionally, more recent than the version that was in effect at the time of baselining the compiler. Depending on the software volume, such testing may range from internal testing to formal validation. The requirement for such testing is automatically satisfied if the compiler is a validated Base Compiler.

As a risk reduction strategy, project managers should consider using validated compilers as early as possible in a project. Maintenance and enhancement revisions of these compilers should be periodically checked for conformance with the language standard by using the ACVC as the test mechanism, so that potential acceptance problems at the time of delivery of software for operational testing are prevented.

10. Compilers for Embedded Targets

Embedded targets pose another challenge to the validation process: such targets may not be able to execute the ACVC tests, due to lack of hardware, in particular of appropriate output devices; the hardware for such targets may not exist yet; the hardware may be so restricted that it cannot support the mandatory features of the Ada language standard. In all these cases, the validation process cannot be applied to the compiler targeted at the embedded system and, hence, no Validation Certificate can be issued for the compiler.

Since it is nevertheless desirable to use compilers conforming to the Ada language standard in developing software for such embedded targets, the proposed DoD policy addresses this issue and arrives at a compromise that alleviates the problem and achieves the objective of conformance to the language standard:

For many embedded systems, the application code is first developed using a "simulated target", i.e., a simulator or an extended hardware configuration that provides more capabilities than the real embedded target system. If the compiler for the simulated target is a (project-)validated compiler, then the compiler for the real embedded target is also regarded as a project-validated compiler, provided that

1. the compiler for the real embedded target is derived from the project-validated compiler for the simulated target, and

2. all mandatory features of the Ada language standard that can be supported by the real target

4. any additions or changes to the run-time support maintain conformance of the execution of application code to the semantics of the Ada language standard.

From the viewpoint of practicality, the compiler for the real embedded target and any run-time modifications are unlikely to deviate from the language standard in an incompatible fashion (as opposed to a subsetting of the run-time support), since the application code translated by the validated compiler for the simulated target, on which the application code is developed and tested, cannot differ significantly from the code produced by the compiler for the real embedded target. Otherwise such earlier testing would be rendered quite useless. Consequently, the conditions enumerated above are sufficient to ensure a maximum of portability and reusability achievable for the application software.

11. The Economics of Compiler Validation

Without doubt, the validation process is expensive for the supplier of an Ada compiler and ultimately for the user of such compilers, since the cost of validation will be reflected in the cost of the compiler to the user. Typical AVF fees for the validation process are $12,000 to $15,000. Suppliers of validated Ada compilers have, however, estimated the total cost of a formal validation in the range from $50,000 to $100,000, due to the internal cost of preparing for the validation. While this estimate seems high, given that the monitoring of results of running the ACVC tests should be accounted for as part of the internal quality assurance for the compiler rather than of validation, a non-trivial effort is spent on administrating the validation process, on conducting the on-site testing, on making sure that the supplier's interpretation of the applicability of individual ACVC tests conforms to the AVO's official position, and on adjusting the compiler in areas where the AVO does not share the supplier's opinion. Extrapolating from these high estimates, the cost of revalidation of a compiler is likely to range from $25,000 to $50,000, provided that neither the compiler nor the ACVC test suite has changed significantly since the previous validation. Even for Registered Derived Compilers, the cost of ascertaining that the compilers indeed are capable of passing all applicable ACVC tests can be substantial to the compiler supplier.

For the user of Ada compilers, the economic benefits of using validated compilers can be considerable. First, the DoD mandate for using validated Ada compilers in MCCR projects makes it impossible to successfully compete for DoD contracts without access to a validated Ada compiler. Second, for the user concerned with porting Ada software between systems, the use of validated compilers is a first measure to reduce the cost of such porting by eliminating the problem with subset or superset compiler implementations. While compliance of compilers to the language standard is not a panacea for solving the porting problem, it eliminates one of the major hurdles encountered in the past with other languages.

12. The Impact of Validation on the Ada Effort

The requirement for validation has been blamed for the long delays in getting Ada compilers to the market place. Historically, the first compilers marketed for comprehensive languages such as COBOL or PL/1 processed only subsets of the respective language, thereby enabling first experiences to be gained with these new languages before compilers for the full languages appeared on the market. This introduction strategy is a two-edged sword, however, since the subset compilers typically remain in use, thus creating significant porting problems. Furthermore, by the time compilers for the full language emerge, programming styles have been established that perpetuate a limited utilization of the full capabilities of the language.

For Ada, DoD consciously decided on a strategy in which only compilers for the full language are acceptable for software development. Within this constraint, the availability of the ACVC has helped the implementors, by providing an extensive and well designed test set, as much as hindering them by forcing them into adherence to minute and sometimes pathological details of the Ada language standard and into the time-consuming process of formal validation.

By the end of 1985, more than 15 compilers were listed by AJPO as formally validated compilers, many of which are hosted on and targeted to a variety of systems. A significant number of additional compilers are expected to be validated in 1986. It is safe to say that the Ada effort is now past any initial delays that may have been caused by the mandate to validate Ada compilers.

The requirement of validation has also led to a situation in which compiler suppliers concentrate first and foremost on the task of obtaining validation status for their compilers, before concentrating on making their products sufficiently efficient and user-friendly to be viable tools in the development of Ada software. This situation must be considered a real danger to the casual public perception of the value of Ada compiler validation, as a validated compiler is by no means guaranteed to be a compiler usable in real application programming. Therefore it cannot be emphasized enough that evaluation of Ada compilers for their respective application domain remains a prime responsibility of the user. Validation can be one decision criterion in this evaluation, but surely not the only one.

Initial concerns about the combinatorial explosion of compiler validations with its cost and probable scarceness of AVF availability should be resolved by the proposed revised validation policies.

On the positive side, the requirement for validation has provided the user with an initial screening of the market place for Ada compilers: since the investment for getting an Ada compiler into a validatable state is

very high, the market place has focused on the viable compiler suppliers likely to provide good products and continued product support. Moreover, the publicized list of validated compilers, and the Validation Summary Reports (see Appendix A), which are publicly available, provide the user with valuable information about the products offered in the market place.

The validation process for Ada has raised the awareness of the suppliers as well as the users for the importance of compliance of a compiler with the programming language standards.

In the long run, DoD's persistence in its "no dialects"-policy for Ada and its validation requirement will have made a significant contribution to the portability of Ada programs and to the potential emergence of an Ada software component market which would be considerably less viable without such a standard. Compared to the cost reductions that can result from these facts, the cost of validation, especially when distributed over a large user base, is a minor expense that can be easily justified by the derived benefits.

APPENDIX A
THE TEN STEPS TO VALIDATION

A.1. Step 1: Self-Testing

The supplier of an Ada compiler must obtain a copy of the ACVC test suite for in-house testing of the compiler. The test suite is distributed by the AVF for a nominal charge covering the cost of distribution. The suppliers compile and execute the test suite using their compilers and, typically, will proceed to step 2 only when compliance of the compilers with the test suite has essentially been established.

A.2. Step 2: Notice of Intent to Validate

The suppliers must notify an AVF that they wish to be scheduled for formal validation, and establish a contract with the AVF for its services. The AVF will schedule validations on a first-come-first-served basis. It is therefore advisable for suppliers to contact an AVF as early as possible, once a target date for full compliance with the test suite can be internally established.

In planning for the validation, there are two dates that are of crucial importance to the suppliers:

; the expiration date of the test suite version that the supplier wishes to use for formal validation; and

; in the case of a revalidation, the expiration date of the existing validation certificate.

As there is generally a 90-day lead-time for an AVF in preparing the formal validation, as well as potential resource constraints of the AVF due to other scheduled validations, suppliers must plan well in advance to complete the validation process before these two crucial dates.

Formal validation testing with an already expired ACVC version is permissible only if a declaration of conformity (see step 4) has been made by the supplier prior to expiration of the ACVC version and no test disputes that are pending are decided against the supplier's position.

A.3. Step 3: Contract Negotiations with the AVF

A formal contract needs to be negotiated between the supplier and the AVF, in which schedules and payments to the AVF are agreed upon. As a major part of the work will be performed by the AVF prior to the on-site testing, advance payments will generally be required.

A.4. Step 4: Declaration of Conformity

The compiler supplier is required to provide a Declaration of Conformity to the AVF. In this declaration, the supplier asserts that the compiler to be validated conforms to the Ada language standard. Implementation-specific information, such as required by Appendix F of ANSI/MIL-STD-1815A, must be supplied to the AVF.

Furthermore, the results of running the ACVC tests on the compiler must be submitted to the AVF for evaluation at least 60-days prior to on-site testing, as well as a list of tests that the supplier believes to be incorrect or inapplicable to the compiler to be validated.

The AVF will carefully evaluate the submitted material, resolve any test disputes and issues in coordination with the supplier, and prepare a draft test report.

A.5. Step 5: Resolution of Test Issues

If the supplier and the AVF disagree on test methods or the applicability of tests, the AVF will refer these issues to the AVO for a binding decision.

The AVO will arrive at a decision within two workweeks and convey this decision to the AVF. The AVO is assisted in this process by the "Fast Reaction Team", a group of Ada language experts.

A.6. Step 6: On-site Testing

The AVF will conduct the conformity testing of the compiler at the location designated by the supplier. This test consists of running all applicable tests of the ACVC test suite. The compiler passes the conformance testing, if all tests are processed according to the language standard. A disposition on the outcome of the on-site testing will, however, not be made until step 8 is completed.

A.7. Step 7: Report Preparation

The AVF will collect and evaluate the results of on-site testing and then issue a draft Validation Summary Report (VSR) within 30 days of the on-site testing.

A.8. Step 8: Review of Draft Report

The supplier, AVO, and AJPO receive the draft VSR for concurrent review and comment. Any comments must be submitted to the AVF within two workweeks of receipt of the draft VSR. The AVF will then forward a final VSR to the AVO within two workweeks. If disagreements on the contents of the VSR arise, the Director of AJPO will decide after hearing arguments.

A.9. Step 9: Approval Review

The AVO will review the VSR for proper reflection of all comments made in Step 8, and then forward the VSR to the Director of AJPO for signature.

A.10. Step 10: Issuing the Validation Certificate

A certificate of conformity, called the "Validation Certificate", will be issued to the compiler supplier by the AJPO when the Director of AJPO has signed the VSR. This certificate lists the compiler and describes the configuration on which the compiler was tested on-site by the AVF. The Validation Certificate and the VSR are then made publicly available. The compiler is added to the list of validated compilers.

APPENDIX B

THE ADA VALIDATION FACILITIES

The following AVF are chartered by AJPO to perform Ada compiler validations. The ACVC test suite can be obtained from any AVF or directly from the Wright-Patterson Ada Validation Facility, which acts as the central maintainer of the test suite.

Wright-Patterson Ada Validation Facility
ASD/SIOL
Wright-Patterson AFB, OH 45433-6503
Tel: (513) 255-4472
Contact: Georgeane Chitwood

Federal Software Testing Center (FSTC)
Office of Software Development & Information Technology
Two Skyline Place, Suite 1100
5203 Leesburg Pike
Falls Church, VA 22041-3467
Tel: (703) 756-6153
Contact: Arnold Johnson

Bureau d'Orientation de la Normalisation en Informatique
Domain de Voluceau-Rocquencourt
B.P. 105
F-78153 Le Chesney, FRANCE
Tel: 33-3-955-2535
Contact: Jacqueline Sidi

IABG-AVF
Industrieanlagen-Betriebsgesellschaft (IABG)
Dept. SZT
Einsteinstrasse
D-8012 Ottobrunn, Fed. Rep. of Germany
Tel: 49-89-6008-3090
Contact: Helmut Hummel

Ministry of Defense (PE)
EQD "Aquila"
Golf Road
Bromley, Kent BR 1 2JB, United Kingdom
Tel: 01-467 2600 (Ext. 6056)
Contact: Kevin E. Phillips

PROGRAMMING WITH ADA[1] -- THE ADA ENVIRONMENT

by

Erhard Ploedereder, Ph.D.

Tartan Laboratories

477 Melwood Ave.

Pittsburgh, PA 15221

U.S.A.

Summary

This paper examines two aspects of using Ada for the implementation of large program systems. First, those elements of the Ada language that are particularly targeted at programming in the large are discussed. Conclusions about appropriate design methodologies that match these language features are presented along with an explanation of some potential problems. Second, an overview of efforts to develop programming support environments for Ada beyond the Ada compilation system is given. The rationale and the scope of on-going standardization work in the area of Ada Programming Support Environments (APSE) is presented.

Preface

A necessary prerequisite for using Ada as the implementation language in a project is the availability of Ada compilers. At the end of 1985 more than 15 Ada compilers had been validated on a variety of host and target systems (1). Hence, at present, there is a significant number of Ada compilers available to prospective users of Ada. The focus for satisfying prerequisites for the successful application of Ada is gradually shifting to the software environments around the Ada compiler.

Given a high-quality compiler, the productivity of software designers and programmers depends to a very large degree on the properties of the software development environment. While this is true for any language, it has been specifically acknowledged in the Ada program: considerable efforts are spent to address the issues and improve the quality of software environments.

In this paper, we concentrate on two aspects of the software environment for Ada: first, we discuss some properties of Ada that impact environment issues and programming in the large. Then we present U.S. Department of Defense (DoD) and industry efforts towards evolving software environments in support of projects using Ada.

1. Program Structure in Ada

Historically, programming languages and their compilers supported the programmers by translating individual modules, but failed to enforce interface conventions across module boundaries. Violations of these conventions were discovered in part at link-time and in part during testing by exploring the causes of incorrect program execution. In some languages, e.g., Pascal, C, and Pearl, the missing facility for consistency checking across module boundaries was added in subsequent language revisions or by language extensions of individual compilers or by tools of the programming environment that analyze the interdependent modules for possible inconsistencies.

The Ada language design has been guided by established principles that facilitate programming in the large (2,3). It supports the structuring of program systems into units, each of which can be compiled separately from other units with which it interfaces. The mechanisms for enforcing the consistency among separately compiled units are an integral part of the language. The rules of the Ada language for consistency checks across compilation unit boundaries are as if all units involved were translated in one monolithic compilation; unit boundaries have no influence on the strength and nature of the checks. Thus, dividing large program

---

[1] Ada is a registered trademark of the United States Government, Ada Joint Program Office

systems into many units according to some structure induced by the applied design methodology carries no penalty in terms of the compiler support for the early detection of errors in the program.

In the Ada language, separately compilable units come in two flavors: A unit can be a specification unit, which describes only the programatic interface presented to users of the unit, or it can be a "body", which implements the details of an associated specification. Separately compilable specification units are called "library units" and their bodies are referred to as "secondary units", as they require the existence of primary specifications that describe their interface presented to other units.

Library units are individual packages, subprograms and tasks; such packages and subprograms may be generic, i.e., they describe an entire family of closely related packages and subprograms. Secondary units are the corresponding package bodies, subprogram bodies and task bodies. Packages are used to group logically related specifications, definitions and declarations. Specifications of packages, subprograms and tasks can also be nested within other units, and their bodies can be segregated into so-called "subunits" that can be separately compiled as secondary units.

Any specification unit can be viewed as a contract between the provider and the user of the services offered by the unit. The rules of the language guarantee that neither the user (i.e., another unit) nor the implementor (i.e., the body for the specification) of such services can violate the programatic interface described by their specification. Moreover, the details of the implementation are hidden from the user of the service, thus guaranteeing that the implementation can be changed without affecting the validity of the programatic interface presented to the user. In more technical terms, interfacing is allowed only to specification units, which contain all the information required for checking the consistency of references that cross unit boundaries. Additional features of the Ada language, such as private types, permit the hiding of data representation defined in the specification of a unit, so that users of the service cannot inappropriately take cognizance of representational details that are to be considered implementation-dependent (but generally are needed by the compiler in translating dependent units).

The described elements of the Ada language, which provide global structure in Ada programs, allow a variety of development strategies. Bottom-up development is supported in the sense that already implemented library units can be used to provide the building stones to construct additional library units and their bodies. Top-down development and stepwise refinement is supported in a dual fashion: first, the separation of specification and body for each library unit makes it possible to postpone the implementation of a unit and base all compilations of dependent units on the specification alone. Second, within secondary units, the implementation of locally declared packages, subprograms and tasks can be separated into subunits without impacting the capability to compile any units dependent on the units whose implementation is thus delayed. It becomes possible to code and compile programs whose underpinnings have been specified but not yet implemented.

Generally, the compilation of a unit cannot depend on a secondary unit. Exceptions to this rule are subunits, which depend on the secondary units within which their specification is provided, and compiler-introduced dependencies on secondary units. The latter may arise for the instantiation of generic units, if their bodies are expanded in place, for calls on inline subprograms, and for optimization-related reasons among units that are submitted in a single compilation.

Within the limitations of these exceptions, the clean separation of the implementation from the specification permits arbitrary replacements of secondary units to be made without affecting any dependent units, since such replacements must be in conformance with their respective specifications. This freedom has significant advantages for a top-down development, since defaulted bodies can be provided for units not yet implemented. With such defaults, the program can be brought to execution as long as the defaulted entities are not referenced in a way relevant for the results of the execution. Later, the defaulted bodies can be gradually replaced by their true implementation with a minimum of recompilation effort.

In practice, software development is often a mixture of top-down design and subsequent bottom-up implementation combined with corrections to the original design. The decribed features of the Ada language are ideally suited to support bottom-up and top-down strategies as well as a mixture of the two approaches.

2. *The Ada Program Library*

In order to achieve consistency checking across compilation boundaries, the compiler must retain information about the separately compiled units. This capability is provided by the use of the "program library" which contains the required information about compiled units.

The language rules require that each unit begins with a indication of all units on which it depends. This specification is called a "context clause". Based on the context clause, the compiler retrieves the stored information about the referenced specification units, uses this information for the compilation at hand, and, for each reference to an entity in such a unit, performs the required consistency checks the ensure the legality of the reference.

The concept of separate compilation requires that all units on which a given unit depends have been compiled prior to the compilation of that unit. Consequently, mutual dependencies of compilation units cannot be accommodated. Since dependencies generally exist only with respect to specification units, it is nevertheless possible that the implementation bodies of two units depend on the specification of the respective other unit. For example, it is not permissible that two specification units, A and B, reference each other; it is however possible that the body for A references the specification of B and vice versa. In terms of a design methodology, this restriction implies that mutually dependent definitions must be provided within a single package. It applies in particular to type definitions in package specifications, since establishing the representational details of types cannot be postponed to the compilation of the package body for reasons of compiler implementation constraints.

During software development, compilation units will be subject to changes, which may affect the validity of other compilation units that depend on the altered unit. Most programming languages leave this aspect of the software development process entirely up to the user or to tools provided by the language environment. The Ada language, in contrast, requires that the compilation system recognizes this potential danger and prevents the occurrence of inconsistencies introduced by a change. The mechanism for doing so is also embedded in the program library. It records the dependencies between compilation units and, upon recompilation of a unit, recognizes the fact that dependent units are now potentially inconsistent and may have to be recompiled.

A further application of the recording of dependency information is that the implementations of the program library are capable of retrieving the exact set of units needed to bring a given main program to execution. By transitively accumulating the units and their bodies on which the main program depends, any superfluous units contained in the library will be omitted from the linked image of the program.

With the requirement for the existence of a program library that tracks dependencies and the effects of changes, the Ada language goes beyond the nature of a mere implementation language. It attempts to address some of the problems that historically have been in the realm of software environment tools, i.e., version and configuration management tools. There can be no doubt that these rules of the Ada language will provide a maximum of safeguards against inadvertent errors in maintaining system consistency in the presence of changes.

The requirements posed by the language standard on the capabilities of the program library in this area are, however, rather rudimentary. They merely require that, after compilation of a unit, any previously compiled unit that is affected by the change to this unit must be treated as if it were as yet uncompiled.

An implementation of the Ada program library that satisfies only the minimal requirements imposed by the language standard is likely to create some difficulties for the users, unless a very rigorous programming discipline is enforced. The reason for such difficulties lies in the above rule which, in a unsophisticated implementation of the Ada library, will cause all compilations to obliterate the results of previous compilations of dependent units. Hence, a minimal change to and recompilation of a specification unit can cause many hours of additional recompilations for units that depend on the changed unit, even though the change may have had no real effect on them. The most pathological example of such a change is the addition of a comment to a specification unit. A compilation system that is not capable of recognizing the irrelevancy of this or other changes on the consistency of the compilation results for dependent units will have to require a recompilation of all dependent units. The undeniable advantage of strict enforcement of consistency by the recompilation rules can be quickly negated by the loss of productivity due to delays caused by such unnecessary recompilations.

As many Ada compilation systems will not possess the sophistication of analyzing changes with regard to their effects on dependent units, the methodology employed in the utilization of these systems for the development of Ada software must compensate for this lack. It must ensure that changes to specification units on which many other units transitively depend are minimized. The same holds, to a lesser extent, for changes to units that have subunits. Fortunately, this constraint is consistent with good software design and development practice that stabilizes central interfaces as soon as possible and performs a rigorous change control on these interfaces. Generally, bottom-up implementation strategies, following a detailed top-down design, will provide the best match with the properties of unsophisticated implementations of the Ada program library mechanisms.

The requirements on the Ada program library do not address the problem of parallel versions of units in multiple configurations of an encompassing program system. The solution to this problem, as it may be provided by environment tools, is somewhat constrained by the necessity to co-exist with the rules of the Ada program library. A typical scenario during software development is that new versions of some compilation units are installed for experimentation purposes. If these new versions have led to recompilations of dependent units, then reverting to the initial state after completion of the experiment is not possible without another recompilation of these dependent units. Short of an Ada program library that is fully integrated into a version and configuration control system, the only alternative for preserving the initial state consists in the creation of a new program library for each such experiment. It is therefore crucial that such creation of program libraries be possible with a minimum of effort and resources and a maximum of sharing with existing libraries. Again, the minimal requirements imposed by the Ada language standard do not address this issue.

Finally, it should be noted that the Ada language rules regarding the program library refer only to the results of compilations, but not to the input to these compilations. That is, the program library need not administrate the Ada source files. It is merely concerned with the internal representation of the compilation units as required by the separate compilation capability, and with the generated object code. It is not necessarily a source control system, nor is it required to support dependencies other than compilation dependencies, such as for example the interrelation between source code and documentation files.

It is entirely possible that the productivity of Ada software implementors will be influenced by the quality and functionality of the Ada program library at least as much as by the quality and speed of the Ada compiler itself. It is therefore of utmost importance that the software design and development methodology and the strategies for version and configuration control be in unison with the capabilities of the employed Ada program library, and that the program library mechanisms integrate well with software environment tools beyond the Ada compiler.

Despite all the caveats expressed in the preceding paragraphs, a reasonably sophisticated implementation of the Ada program library that supports functionality beyond the minimal requirements imposed by the Ada language standard can be an extremely powerful tool. In large application systems, errors that are caused by minor interface inconsistencies are very difficult to locate. The required capabilities of the program library prevent the occurrence of this class of errors. With only a moderate addition of functionality to the Ada program library, tedious and traditionally error-prone tasks, such as the recompilation of changed sources and ensuing recompilation of dependent but unchanged sources, can be totally automated and performed without errors, since the required information is directly derived by the compiler from dependency information in the program library.

3. *Programming Support Environments*

For software engineers and programmers to be effective, the provision of an Ada compilation system alone is clearly not sufficient. They require additional tools, such as editors, debuggers, version and configuration management tools, network file-transfer tools, project management tools, and so on. The collection of these tools is generally referred to as a "Programming Support Environment (PSE)".

Some of the tools in a PSE are heavily language-dependent, for example the compilers, syntax-directed editors, program analyzers, symbolic debuggers, performance monitoring tools, etc. These tools need to be developed for any implementation language. Other tools are language-independent, such as file-transfer mechanisms, project administration tools, documentation systems, test harnesses, etc. Where available, these tools can be applied in a project regardless of the chosen implementation language.

Since the early stages of the Ada effort, considerable attention has been focused on the PSE for Ada and, in the process, on issues of language-dependent and -independent environment support in general, because the state of the art in this area is in its infancy despite its recognized importance. Unfortunately, this attention has led some observers to the misconception that Ada requires substantially more environment support than other languages or, worse, cannot be used at all without a complete environment specifically developed for Ada.

In reality, Ada has been used as a focal point of plans for improvements in software engineering and PSE technology, which are direly needed regardless of the choice of implementation language. It could even be argued that Ada may require less environment support than other languages, due to its high degree of compile-time error checking, its enforcement of implementation discipline, and the environmental aspects of the program library. There is certainly little reason to believe that Ada could not be supported in more traditional environment settings. Currently, the majority of commercially available Ada compilation systems are not embedded in an Ada-specific PSE, but are integrated into the standard PSE available on the respective host systems.

3.1. The DoD Requirement Catalogues

In early 1978, first efforts were made within U.S. DoD to arrive at a concept for the development of the environment support for Ada. A set of initial ideas were first collected in what became known as the SANDMAN catalogue, which was never published. By late 1978, it was consolidated into a requirement catalogue, named PEBBLEMAN, in which the desired functionality and cooperation of various tools were described. A revised version of PEBBLEMAN was published in 1979 (4). In February 1980, a further document, STONEMAN, was produced; it became one of the most cited references regarding Ada environments (5).

In addition to posing requirements for the desired tools in a PSE, STONEMAN introduced a model for "Ada Programming Support Environments (APSE)", which addressed both the issues of tool integration and of portability of the entire APSE as well as of individual tools or tool-sets. The term "APSE" has since become a synonym for the concept of integrated toolsets for Ada, as opposed to the so-called "tool-box" approach in which a set of independent tools is provided to the user.

Part of the motivation behind the STONEMAN model was the recognition that the general immaturity of current PSE implementations is not caused so much by the non-existence of powerful tools as by the missing capability to bring existing tools together on a single host system and integrate them with each other to form a coherent and efficient PSE. Consequently, the development of a PSE becomes unnecessarily expensive, as many tools are reimplemented although similar and, quite possibly, better tools are already available in other environments.

One might surmise that the enhanced portability of tools written in Ada would solve the portability problem. However, while the Ada language standard provides portability advantages in many areas, it must be recognized that most tools require a significant amount of interfacing with the host operating system. The Ada standard, which is primarily intended for the generation of application code for arbitrary target systems, could not justifiably prescribe the details of those language features that are intimately linked to operating system interfaces. Moreover, the requirements of application code on operating system services have been recognized to be quite different from those of PSE. This is due partially to the different problem domains and partially to the dynamic nature of evolving PSE as opposed to the relatively static nature of operational application systems.

Consequently, some standard interfaces suitable for application programming might be quite inappropriate for PSE development. Given that the Ada language leaves the details of operating system interfaces, such as calls on the file management or on terminal IO services, largely implementation-dependent, the porting of tools written in Ada will nevertheless have to contend with modifications of these host-dependent portions of the software.

The STONEMAN model postulates a system architecture in which the host dependencies and the tool intercommunication are encapsulated in a Kernel APSE (KAPSE). The individual tools of the APSE are built on top of the KAPSE services. Since the interfaces offered by the KAPSE are conceived to be host-independent, and since the Ada language goes to great length in facilitating the portability of Ada software, tools could be written in Ada to be portable among different hosts offering the same KAPSE services. Port-

ing of an entire APSE to a new host consists of re-implementing the KAPSE on the new host. Furthermore, individual APSE tools could be transferred to another APSE as long as the tool intercommunication interfaces needed by these tools were provided by this APSE. Some tools are so heavily dependent on interfaces with other tools that it is unreasonable to expect that all such interfaces are provided on each KAPSE. These tools cannot be ported individually; they form an tightly integrated tool-set. By relying only on the common mechanisms for tool communication provided by the KAPSE, but not on the specific details of the communicated information, such combined tool-sets could equally be ported to a new APSE.

STONEMAN established a set of requirements for the services that must be provided by the KAPSE in order to reach the described goals. These requirements relate in particular to the data administration and communication interfaces needed by tools, and to the run-time system that enables the execution of Ada programs. STONEMAN adopts the paradigm of distinguishing host and target systems. It postulates that software development, in particular for embedded targets, needs to take place on a host system sufficiently powerful to accommodate the various tools required for supporting the development and maintenance of software throughout its life-cycle.

Among the many conceivable tools of an APSE, STONEMAN identifies a minimal set perceived to be necessary to make an APSE an effective tool for software developers and maintainers. This set was designated as the Minimal APSE (MAPSE). It comprises the Ada compilers, linkers and loaders, simple static program analyzers and debuggers, text editors and pretty-printers, a file and configuration management system, and the command interpreter.

The STONEMAN principles were readily accepted by the majority of efforts concerned with the development of a PSE for Ada. In particular, the emphasis of STONEMAN on the provision of better data administration capabilities than offered by the file management of traditional operating systems has been reflected in almost all major PSE developments. While some of the details of the STONEMAN requirement catalogue may require revisions in the light of experience gained since 1980, the fundamental principles are still valid contributions to the area of PSE design.

3.2. APSE Implementations

Within U.S. DoD, two major developments of Ada Programming Support Environments were procured: In 1980, the U.S. Army initiated the development of the Ada Language System (ALS) with SofTech, Inc., as the lead contractor (6); in 1981/82 the U.S. Air Force contracted with Intermetrics, Inc., for the development of the Ada Integrated Environment (AIE) (7,8). Initial plans called for the ALS to be an interim tool-set for the introduction of Ada, while the AIE was intended to be a STONEMAN-conforming integrated environment and to eventually become a standard DoD Ada environment.

The developers of the ALS adopted many of the STONEMAN principles in their implementation strategy, such as utilization of a KAPSE-like kernel to facilitate the porting of the ALS, which has been developed on a VAX/VMS host system. In December 1983, the first version of the ALS was made available to prospective users. The self-hosted Ada compiler of the ALS was validated in December 1984. The ALS comprises a set of about 75 tools to be used in software development and maintenance. The development of the AIE, whose initial design promised a superior integrated environment, encountered serious funding problems; at present, it is extremely doubtful that a comprehensive AIE will become available in the foreseeable future.

The U.S. Navy decided in early 1985 to base their Ada environment efforts on the ALS and to enhance this environment by additional tools and by code generation capabilities for the prevalent instruction set architectures in use by the Navy. This extended ALS has been named "ALS/N" and is expected to become available in early 1989.

The German Ministry of Defense, Bundesamt fuer Wehrtechnik und Beschaffung, began the procurement of components of an Ada software environment, named SPERBER (Standardisiertes Programm-Erstellungssystem fuer den Ruestungsbereich) in 1979 (9). Major efforts have been directed at developing Ada compilers, debuggers, and a program development data base system. The first two compilers were validated in November 1984. The British Ministry of Defense in cooperation with British industry co-financed several design efforts towards the development of software environments for Ada (10, 11). The Commission of European Communities, under its multi-annual program to advance the European software technology in the commercial sector as well as under the ESPRIT program, has co-financed several multi-national projects developing Ada programming support environments, most notably the PAPS project (11).

Commercial suppliers have been somewhat reluctant to embark on a course of providing integrated Ada Programming Support Environments, recognizing that such integrated solutions, while desirable in principle, constitute a truly major capital investment. Moreover, customers who have built a considerable wealth of software to support their in-house software development are concerned with preserving the usefulness of this software for future development projects using Ada as the implementation language. The challenge to commercial suppliers is to provide integrated environments that nevertheless allow the inclusion or easy transitioning of existing tools.

Apart from ALS, which is also commercially marketed by SofTech, Inc., only the Ada Development Environment (ADE), marketed by ROLM Corporation (12), and the Rational Environment, marketed by Rational, can be regarded as largely integrated Ada Programming Support Environments. The Rational Environment takes a quite unique approach: its host system has been specifically designed for the development and execution of Ada programs. The entire environment is exclusively centered around Ada. Many of the language concepts are immediately reflected in the concepts of the environment whose command language is Ada. Other commercial suppliers of Ada compilation systems have taken the path of embedding their systems into the environments offered by existing operating systems, so that users could continue to utilize the tools which which they are most familiar.

4. *Environment Standardization Efforts*

The more the software development process is assisted by tools of a programming environment, the more difficult it becomes to transition software developed in one environment to a different environment. The current practice in military procurement of mission-critical software is that this software is developed by contractors but maintained in military maintenance centers. Just as a proliferation of implementation languages raises the cost of such maintenance considerably, so does a proliferation of software environments needed to maintain the software even in a single language. It is therefore in the interest of DoD to minimize the proliferation of environments in its maintenance centers.

Here, DoD is faced with a dilemma: while language research was advanced enough to embark on the standardization of a single language, Ada, today's state of the art in environments is much less mature. Therefore rigorous standardization on a single environment may be ill advised, as considerable advances in the environment technology can be expected to occur in the next decades.

A compromise can be found within the framework of the STONEMAN model. If commonality of KAPSE implementations were advanced by standardization at this much lower and less ambitious level, tools used in application development could be transitioned into existing maintenance environments, thus reducing the number of necessary environments while, at the same time, continuously enhancing the support provided of these environments.

4.1. The Kapse Interface Team (KIT)

When it became apparent that there would be two competing designs of KAPSE interfaces for the ALS and the AIE respectively, a Memorandum of Agreement was signed in January 1982 between the U.S. Army, Air Force, and Navy to work towards establishing commonality of the KAPSE interfaces in DoD environments (13). Under the lead of the U.S. Navy, the KAPSE Interface Team (KIT) was created and chartered with this task. The KIT is assisted by an advisory group of software environment experts from industry and academia with international representation, the KITIA (KIT - Industry and Academia). The objective of KIT/KITIA was set to establish requirements for the interoperability and transportability of tools among APSE and to subsequently develop guidelines and conventions for achieving these requirements with the ultimate goal of evolving standards in this area (14).

KIT/KITIA has been meeting quarterly since 1982. The results of its deliberations are contained in periodically published reports (14). Among its most relevant products are the "Requirements and Design Criteria for the Common APSE Interface Set (CAIS)" (15), and the proposed "Military Standard Common APSE Interface Set" (16), which is an initial set of KAPSE-like interfaces for the encapsulation of host dependencies. At this time, the later document is being reviewed by the DoD services for adoption as a military standard within DoD.

While the initial motivation of achieving commonality between ALS and AIE has decreased, due to the lessening importance of the AIE, the KIT/KITIA effort has been recognized as an important contribution to ad-

vance the knowledge in the area of KAPSE-like interfaces and of issues of tool portability and interoperability. It also creates a forum for information exchange between DoD and industrial efforts, thereby preserving the opportunity to prevent a complete divergence of these efforts.

4.2. The Common APSE Interface Set (CAIS)

In late 1982, a KIT working group began examining the ALS and AIE KAPSE interfaces for areas of commonality and of divergence in order to establish a common set of interfaces that could be supported by both KAPSE designs. In March 1983, this group was joined by several KITIA members to form a group that later became known as the CAISWG (CAIS working group). With the participation of the ALS and AIE designers, this group began developing the specification for a set of common interfaces deemed important for the portability of tools. A crucial design decision was made in mid-1983 to pursue an interface set that was not constrained by the current environment efforts of ALS and AIE, although much of the practical experience of these and other similar efforts influenced the choice of interfaces.

The main goals of the CAISWG were to develop interfaces based on a simple, yet powerful and extendible, model, to apply uniform concepts throughout the design of the interfaces, and to cover those interfaces that are most crucial for the portability of many tools. A first public review of the concepts of the CAIS took place in September 1983. Various revisions were produced and publicly reviewed until, in January 1985, the final document was delivered to AJPO as a proposed military standard.

The CAIS in its present form contains interfaces for the administration of files, their interrelations and attributes, for process administration, and for terminal IO targeted at three standard kinds of terminals. Some other utilities frequently used by tools are also present. The CAIS design for the file administration has departed from the traditional view of hierarchical file systems and instead administers files by their interrelations. In this regard, the CAIS cautiously joins a trend that has been apparent in almost all recent designs of PSE. Special attention has been paid to security aspects, so that the CAIS would not be in conflict with requirements posed by DoD (17). The concepts and the set of interfaces provided by the CAIS are open-ended. It is expected that additional interfaces will be added to the CAIS and that the existing global concepts are sufficiently flexible to accommodate a large variety of such extensions.

It was realized that the work of the CAISWG could be only a first step in defining a basic set of uniform interfaces. The U.S. Navy contracted with SofTech, Inc., in 1986 for the further enhancement of the CAIS beyond this initial set and for a pilot-implementation of the CAIS to validate its usefulness and efficiency. Other pilot-implementations of major portions of the CAIS have been undertaken by TRW, Inc., under government contract, and by Gould, Inc. and MITRE Corporation, as internally financed projects.

5. Future Benefits of APSE

As various studies have shown, the demand for application software is rising exponentially over time, while the workforce engaged in producing this software is growing at a slow linear rate. Already the demand is far beyond the production capability. In addition, the complexity of the software has increased considerably, making the production of quality software more and more difficult and time-consuming. Without substantial improvements in the software development process, industry will not be capable to meet the increasing demand nor will it master the growing complexity of this process in the future.

Four factors can have a major impact on ameliorating the current situation:

; Application of better methodologies: Ada is a first step to improve the methodological basis at the implementation level. Clearly, better methodologies or increased application of already available methodical approaches for requirement analysis and specification are necessary as well. Almost certainly, these approaches will be supported by software tool-sets. APSE can provide the vehicle for a wider penetration of these tool-sets and associated methodologies.

; Provision of better tools: Today the portability and integration problems of tools lead to a deadlock situation. For lack of portability, the development of truly sophisticated tools is expensive and commercially risky; it therefore remains largely in the realm of proto-types developed in academia. For lack of commercially available tools, software developers are forced to expend their resources in the duplicating development of tool support, which, because of the resulting financial constraints, continues to be of low quality. Improvements in portability and integration, as possibly provided among APSE, can release resources for the development of more advanced

and better tools as well as widen the commercial market for such tools.

; Use of standard components: similar to the hardware manufacturers, software producers will increasingly have to rely on reusing pre-fabricated components in order to meet the rapidly expanding volume of the software demand. Within the framework of an application-oriented implementation language, whose major goal is the enhancement of software portability, Ada is an almost ideal breeding ground for such standard components.

; Automated generation of customized components: a leading edge in hardware research is concentrating on the development of systems for the fast production of customized components. A similar approach can be expected to emerge in the software field as well. In fact, in some specialized areas, e.g., in compiler construction, the automated generation of components that are customized to particular languages or target architectures has already been successfully applied in commercial products. Again, APSE may provide the means for a wide distribution of such software-generating tools.

Of all these factors, the last two are likely to be the most important ones. As long as application software is developed from scratch, whatever improvements in tool support can be provided to the software developers will increase their productivity only by some constant multiplying factor. Significant as this factor may be, it will not be sufficient to catch up with the exponentially developing software demand in the long run. The only hope to match the ascending curve of demand consists in an ever increasing reduction of the amount of work to be performed to produce products. Only the utilization and increasing availability of standard components or of automated generation of customized components can lead to this reduction. Ada and APSE can play a major role in meeting these challenges, in particular, if the promise of increased portability of tools and software components among partially standardized APSE can be realized.

6. *References*

(1) Ada Information Clearinghouse, "Validated Ada Compilers", AdaIC, 3D139 (1211 Fern St., C-107), The Pentagon, Washington, November 1983.

(2) U.S. Government, "Reference Manual for the Ada Programming Language", ANSI/MIL-STD-1815A, U.S. Government Printing Office, Washington, February 1983

(3) Honeywell, Alsys, "Rationale for the Design of the Ada Programming Language", Draft for editorial review, Honeywell, Minneapolis, January 1984

(4) United States Department of Defense, "PEBBLEMAN revised - Requirements for the Programming Environment for the Common High Order Language ", January 1979

(5) United States Department of Defense, "Requirements for Ada Programming Support Environments - STONEMAN", February 1980

(6) SofTech, Inc., "ALS Specification", November 1983

(7) Intermetrics, Inc.,"System Specification for Ada Integrated Environment", November 1982

(8) Bray, G., "AIE Support for Management of Embedded Computer Projects", Ada Letters, Vol. II, Number 1, pp. 33-49, August 1982

(9) Ploedereder, Erhard, "Project SPERBER - Background, Status, Future Plans", Ada Letters, Vol. III, Number 4, pp. 92-98, February 1984

(10) Department of Industry, "U.K. Ada Study, Final Technical Report", June 1981.

(11) Bevan, S. et al. "Investigation into the Differences Between PAPS and M-Chapse", unpublished paper, December 1983.

(12) Elliott J.K., Klein D.M., Williams J.S., "APSE Tools - ROLM's experience", in: Teller J. (ed.) "Proceedings of the Third Joint Ada Europe/AdaTEC Conference", Brussels, 26-28 June 1984, Cambridge University Press, 1984.

(13) Memorandum of Agreement, published in (14), Volume III, pp 3B 18-19

(14) Patricia Oberndorf, "Kernel Ada Programming Support Environment (KAPSE) Interface Team Public Report", Naval Ocean Systems Center, San Diego.

Volume I, NOSC Report TD-209, NTIS AD A115 590, April 1982
Volume II, NOSC Report TD-552, NTIS AD A123 136, October 1982
Volume III, NOSC Report TD-552, NTIS AD A141 576, October 1983
Volume IV, NOSC Report TD-552, April 1984
Volume V, NOSC Report TD-552, August 1985

(15) KIT/KITIA, "DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS)", prepared by KIT/KITIA for the Ada Joint Program Office, September 1985

(16) U.S. Government, Ada Joint Program Office, "Military Standard Common APSE Interface Set (CAIS)", Proposed MIL-STD-CAIS, NTIS AD 157-587, January 1985

(17) United States Department of Defense,"Trusted Computer System Evaluation Criteria", Computer Security Center, Fort Meade, Maryland, CSC-STD-001-83, August 1983.

PRACTICAL EXPERIENCES OF THE ADA® LANGUAGE FOR REAL-TIME EMBEDDED SYSTEMS DEVELOPMENT FOR THE DEFENCE-RELATED MARKET

by

Dr Mel Selwood
Plessey UK Limited
Abbey Works
Titchfield, Fareham
Hampshire PO14 4QA
UK

Summary

This paper describes some of the experiences gained to-date from an Ada research programme, undertaken within the Plessey Company in the U.K., by the author and his team. This programme is investigating the cost-effective and beneficial introduction of the Ada language for Defence-related (mostly real-time) software applications. Particular emphasis is placed upon minimising the risks and maximising the benefits for large and / or embedded microprocessor-based systems. Within the context of this largely practical work programme, the paper identifies a number of key concerns within the team (and it is suggested within industry at large) in making the transition to Ada. Also, some suggestions for improving the application of current Ada compilers and tools is provided to the vendors of these products.

Introduction

Clearly, in order to obtain the longer term benefits claimed for the Ada language, it is first necessary for any commercial organisation to be able to bid for and implement Ada projects both profitably and at minimum risk. This can only be done with confidence (at least for fixed-price contracts) when that organisation has already implemented "representative" Ada projects, so that it can draw upon real experience of the technical issues involved. Only then can it, for example, reliably establish criteria for estimating project timescales and costs, and define appropriate standards and procedures for Ada-based developments.

In general, for large and complex (especially real-time embedded micro-processor based) systems, for which Ada is intended, the risk of using Ada ahead of gaining such real experience may well be too great. Further, although the introduction of Ada may go hand-in-hand with improved software engineering practices, it would be inappropriate to ignore the often large existing investment in non-Ada software and systems products (and the associated manpower skills, working practices, equipment etc.) in favour of some totally new, unproven development scenario.

Instead, the situation demands an optimum (transition) solution for which on the one-hand changes to the status quo are minimised, while on the other hand the potential benefits of Ada and associated development methods are maximised.

This paper reports on some of the practical experiences gained to-date from an Ada research programme which was set up to address this problem.

Ada Research Programme

The programme was set up as a 'virtual' project, involving a hybrid mix of practical study activities and real Ada software developments. This project is ongoing and involves multiple study teams with distributed interests, thereby ensuring a broad approach to the problem.

Given that the coding phase of projects typically equates to only 10% of the overall development effort, it is clear that the real value of Ada will come not from the direct characteristics of the language itself but from the catalytic (secondary) influence upon the software engineering methods employed. Thus, the following issues are being examined:

i. the appropriate time-frame for introducing Ada,
ii. the impact of Ada upon the overall development lifecycle,
iii. the necessary standards and procedures (Project Management Baseline),
iv. the anticipated effort / cost - time profile for Ada projects,
v. constraints upon the design and implementation of products, and their expected characteristics,
vi. operational aspects (eg appropriate development environment).

® Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

Within this context, the programme focusses upon the production of Ada demonstrators for gaining practical experience of Ada, to demonstrate representative Ada-based software systems actually working, and to allow ancillary studies eg into code size and performance issues. The work is complemented by a series of across-application studies to check the consistency of the results and their applicability to varied applications.

Three demonstrators are currently being worked upon:

A Digital Telephone Exchange Demonstrator

This system implements the complex multi-tasking activities associated with setting up, supporting, and terminating, one or more concurrent two-party telephone calls from Digital Voice Terminals (DVTs). The system supports a variety of facilities eg system configuration, abbreviated dialling, diversion of calls, call pre-emption etc.

The system is a conversion of an existing product which was developed using the "Modular Approach to Software Construction, Operation and Test (Mascot) [1]", and implemented using the Coral 66 language. The majority of the new system preserves the existing Mascot design (excluding the Mascot 'machine' concept, which supports primitives such as operations on control queues for synchronising access to shared data areas). The required system functions were then manually re-implemented using the full features of the Ada language. In contrast, the man-machine interface (MMI) sub-system is both newly designed and implemented and this serves as a test case for studies into Ada design methods.

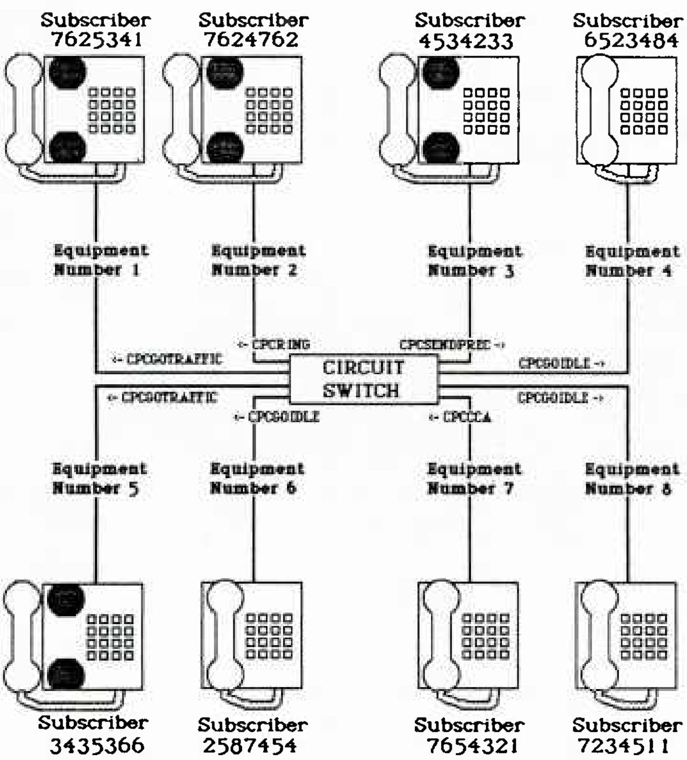


Figure 1 - Typical Screen Snapshot

The demonstrator involves three platforms:

i. an entirely DEC Vax-based Ada system with software emulating the function of the real (DVT) hardware - a typical screen snapshot is shown in figure 1.
ii. the Vax-based Ada system linked via an RS232 interface to an existing hardware rig supporting real DVTs,
iii. the Ada software at ii. re-ported to run on an Intel 80286-based target.

A Sonar System demonstrator

This system is based upon an existing Pascal implementation and demonstrates a typical naval surface ship sonar data processing and colour display function. It presents both Active and Broadband Passive sonar data in grey-scale format, updated in real-time, together with automatic target detection and tracking functions. Operator interaction is via 'soft' keyboards presented on the sonar data displays, and manipulated using a special purpose five-button keypad. A sonar cursor is also provided, controlled from a rolling ball.

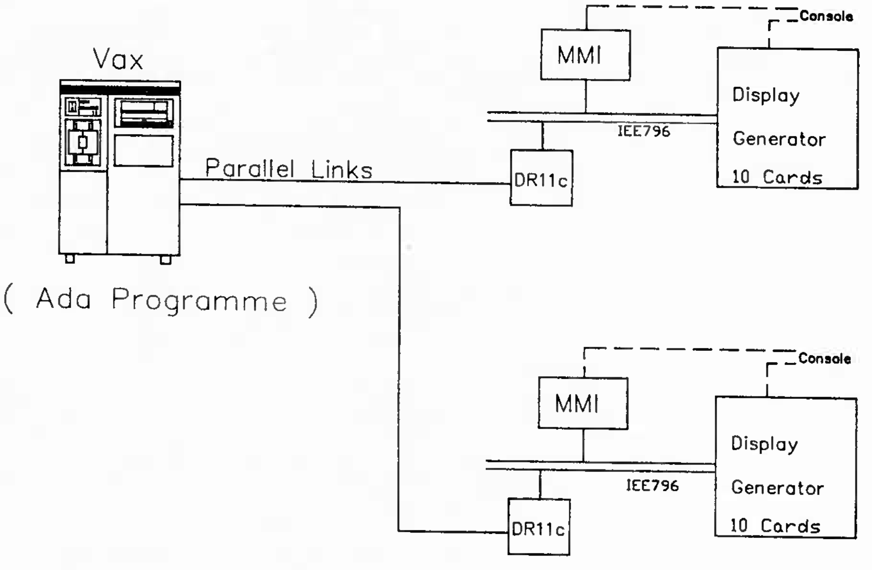


Figure 2 - Platform 1 Sonar Demonstrator

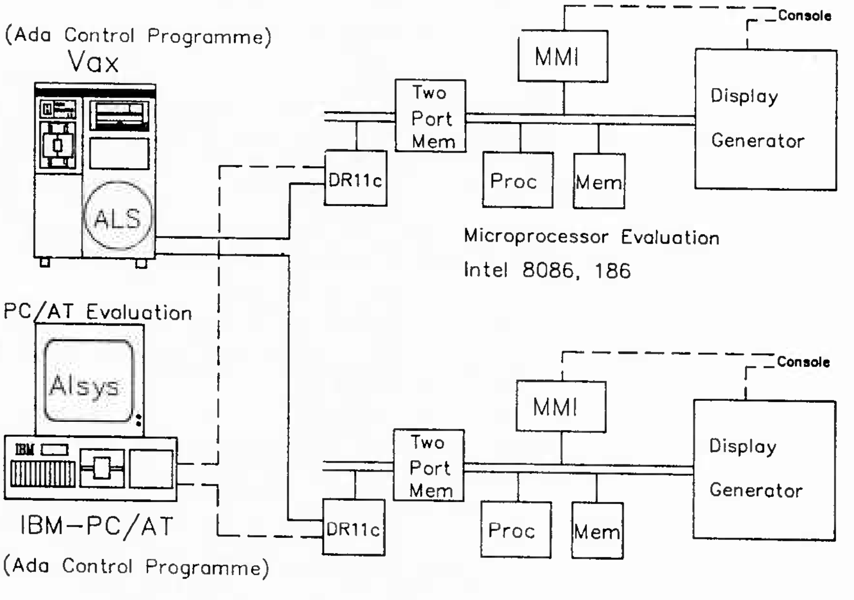


Figure 3 - Platform 2 Sonar Demonstrator

There are two platforms (figures 2 and 3). In the first the Ada software runs in a Vax which is directly coupled via twin parallel interfaces to two IEE796 microprocessor busses, on which are situated multi-plane colour graphics display generators and man-machine interface (MMI) cards. All intelligence is deliberately removed from the IEE796 cards, and accordingly the Vax-based Ada software deals with the display and MMI hardware at the lowest level of bit manipulation.

In the second platform, the majority of the demonstrator is reconfigured for a multiple Intel 8086 microprocessor system whilst the remaining Ada control module runs in either Vax or IBM PC-AT machines.

An Engine Monitoring System demonstrator

This demonstrator is based on an existing product [2] which performs a range of engine life count calculations, incident / exceedance monitoring and vibration analysis, for the Rolls Royce Pegasus engine. It provides a platform for investigating the use of Ada in high performance-critical applications; the majority of complex algorithmic calculations being carried out in real-time. The initial investigations involve re-implementing in Ada that software which is used to assess the low cycle fatigue damage on the major components of the engine.

There are two phases to the work. The first involves an examination of the existing design and Pascal implementation, and re-implementing the system in Ada to run on a Vax. In the second stage the system is re-targetted to run in the real Motorola 68000-based hardware configuration.

Results and Discussion

Digital Telephone Exchange Demonstrator

Development of this demonstrator commenced using the Telesoft Ada compiler V2.1, and later the Karlsruhe Ada compiler V1.1. However, the performance of these products was below expectation and a change was made to the newly arrived DEC Ada compiler V1.0, with which the implementation of the first and second (Vax-based) platforms were satisfactorily completed.

For the third platform, which is still under development, involving an Intel 80286 embedded microprocessor target, it is planned to use the Verdix/VADS system (running under VMS) since the DEC Ada compiler does not currently support code generation for non-DEC microprocessor targets. This work should allow comparison between the DEC and Verdix products.

The following list identifies some of the operational issues which have been raised by the demonstrator experience:

i. The security and robustness of the Ada library management facilities.
ii. The ease with which software developed using one compiler can be recompiled under another.
iii. The robustness of the compiler, and the existence of any implementation constraints.
iv. The development machine resources required.
v. The speed of compilation.
vi. The extent of interference by the operating system in the execution of the Ada software.

Thus, taking point v. as an example: for both the Telesoft and Karlsruhe Ada compilers, in the particular application used in the demonstrator exercise, the CPU time required to build the first demonstr-ator platform (before the system was fully coded) was in excess of two hours - the elapsed time being substantially longer. Considering only the MMI sub-system (ca. 25% of the total system), this meant that the "(re-) build and run - analyse and debug" development iteration could be performed on average two or at most three times per normal working day (depending on the extent of re-compilation required).

When the DEC Ada compiler was used in the same application, the <u>entire</u> system was built in 16 minutes CPU time (typically two hours elapsed time) on a Vax-11/785 with 8 Mbytes main memory. This turn-round time compared favourably with experiences of traditional languages eg Coral, and led to much greater overall productivity.

The platform 1 software comprises some 40 packages, including 34 Ada tasks, and is implemented in approximately 25,000 lines of code (including comments). This represents a substantial working example of a complex Ada system, albeit a Vax-based implementation.

Sonar Demonstrator

The first (Vax-based) platform was implemented using both the Karlsruhe and DEC Ada compilers. In transferring to the DEC Ada compiler it was noted that successful re-compilation and build occurred without needing any code changes at all. Despite the inevitable differences in run-time characteristics, this is an optimistic sign for portability of Ada across different projects.

The first platform is implemented in approximately 8,000 lines of code (including comments). This compares with approximately 5,500 lines of Pascal code in the original implementation. However, this smaller size can be attributed to the reduced number of facilities for error recovery and reduced program robustness, rather than to any verboseness of Ada (except where enforced by the strong typing features of the language).

The speed of progression of the Sonar 'ping-front' from the bottom to the top of the display screen provides a simple measure of the net relative performance of the software. Such measurements show that, in the applications used, for single-threaded (non multi-tasking) versions of the platform, the DEC Ada-based system slightly outperforms the DEC Pascal version. An extended multi-tasking DEC Ada implementation is showing comparable performance to its single-threaded counter-part.

For the second platform (multi-Intel 8086 microprocessor configuration) the SofTech ALS system is being used. However, to meet the operational requirements of the system, the target hardware is not based on standard Intel boards. Accordingly, certain non-trivial problems are involved:

i. Support for the different hardware components employed.
ii. The size of the run-time support (RTS) system - this currently exceeds the amount of the processor card on-board memory.
iii. Alterations to the standard RTS to cater for the different configuration of the target hardware.

This work is ongoing but is already highlighting a number of important issues eg the impact on validation.

Engine Monitoring System (EMS) Demonstrator

In the past for this application the Structured Analysis / Structured Design method [4,5] has been used. For Pascal, this has often involved considerable pre-implementation 'engineering' of the design. However, for Ada the information hiding features (primarily) appear to permit a more optimal mapping with consequential improvements in the software structure. This area of investigation is still at an early stage, but the initial results look encouraging.

Comparing functionally equivalent single threaded (non multi-tasking) Vax-based versions of the software implemented respectively in (DEC) Pascal and (DEC) Ada, shows a decreased run-time performance in the latter case, contrary to the results from the Sonar Ada platform. A number of implementation changes to the Ada version have been made in an attempt to explain this difference, but so far these have not substantially altered the results.

The reduced Ada performance for this application may be due to the significantly higher degree of numerical processing involved, and this is being investigated. Work is progressing on the implementation of a driver/display unit (to display the low cycle fatigue results) for which no such numerical calculations are involved. This should provide an opportunity for further comparison.

For the second stage of the work, it is planned to use the Verdix/VADS cross-development system and this should provide both size and performance data for the 68000-based target configuration in the near future.

Ancillary studies

Apart from the demonstrator projects, a number of additional studies are being carried out in the following key areas:

The design of Ada-based systems: A number of methods eg Structured Analysis / Structured Design [4,5], Mascot [1], the Structured Systems Analysis and Design Method (LSDM/SSADM) [6], and object-oriented approaches [7,8] are being reviewed.

In the design and implementation of the MMI sub-system for the Digital Telephone Exchange Demonstrator, using a Mascot-like approach, it was found that for example:

i. Significant effort was needed to design and efficiently package the data types and objects (Mascot provides insufficient support).
ii. Packaging structures were initially adopted which were subsequently found to be non-optimal. Thus, while undesirable sharing of data objects was avoided, the first implementation required excessive sharing of data type definitions.
iii. Although the strong typing of Ada generally led to a much more straight-forward mapping between the design and code, this was at the expense of some awkwardness in the processing of the data.
iv. Using 'with' alone, rather than 'with' and 'use', for referencing other packages was found to be much clearer and less error-prone (for this large scale development). This contrasts with the (implied) recommendations of most Ada textbooks whose examples are rather simple.

The following points were also observed:

Mascot segregates processing units (activities) and intercommunication data areas (IDAs), the access procedures for which encapsulate the more complex inter-task communication and synchronis-ation aspects. This is useful when the implementation teams are of mixed ability. In Ada, inter-task communication is an implicit part of all of the applications software. The full impact of this upon large systems developments has yet to be established.

A direct mapping of a Mascot design to Ada usually leads to all activities being implemented as (active) Ada tasks. However, since Ada assumes a synchronous tasking model (rendezvous) the implementation of the IDAs leads to two possibilities: (1) treating them as decoupled (asynchronous) inter-task communication mechanisms, and hence coding them as (passive) Ada tasks, or (2) effecting a synchronous inter-task communication by means of a rendezvous. In the first (more general) case, the overall system performance depends even more heavily upon the efficiency of Ada tasking.

It is important in defining the application boundary for an Ada task, to bear in mind that it is not possible to alter the priority of an Ada task dynamically (at run-time). Thus, care has to be taken not to group functionally related processing activities within a single task if the functions are inherently not of equivalent priority.

The MMI sub-system has now been re-designed and re-implemented based upon the findings of the review into this and other methods. This serves as a model example from which a reasonably optimal design and implement-ation code of practice is being derived for future applications.

Program testing: Most Ada compiler vendors are supplying symbolic debuggers for use within the program debugging stage of software development. However, most Defence-related projects involve a high degree of stringent testing (verification and validation against the requirements and design) and this aspect appears to be receiving scant commercial attention.

Accordingly, the suitability of using commercial symbolic debug facilities as the basis of a more sophisticated test harness is being investigated. To-date a prototype test tool has been produced which involves lexical analysis of the software under test and the automated production of command files to drive the DEC symbolic debugger. Further work is being carried out to investigate the representation of the complex real-time behaviour of Ada systems by advanced graphical means.

During this work numerous instances have been encountered of having to gather data about the software under test which must clearly have already been obtained during the course of Ada compilation. However, this information is not made externally visible by the compiler. It is felt that there is immense scope for Ada compiler suppliers to collaborate with industry to help overcome this sort of problem.

Configuration Management: Because of the inherent complexity of most Defence-related programmes, and the fact that they require multiple development teams, strong emphasis is placed on the need for efficient tool-based configuration management methods to support Ada projects. Of particular interest is support for software re-use across projects and in devising suitable schemes for linking existing or future configuration management databases to Ada system build facilities.

Current investigations are looking at the use of the DEC, SofTech and Verdix products for this purpose. The results are expected to form the basis of a future paper.

Run-time support: The provision of efficient run time support for embedded microprocessor-based Ada systems is crucial to the use of Ada for such applications. To-date studies in this area have been frustrated by the frequent lack of detailed data from the compiler vendors about the characteristics of the run-time support systems to be supplied. Again, this is a problem area which could benefit from further collaboration between the Compiler suppliers and Industry. Some of the issues of importance are:

i. the functionality, size and performance of the RTS, and the 'hooks' provided for use by applications software,
ii. the advantages and disadvantages (eg for portability) of using proprietary RTS systems eg Intel's iRMX, and the Hunter & Ready VRTX system, as well as Ada specific products,
iii. the interaction between the underlying Ada compiler technology and the RTS system in the context of the often non-standard hardware configurations used in embedded microprocessor applications.

At the present time software that can be used to 'bench-mark' the commercially supplied run-time support systems is being developed.

Conclusions

The research programme being carried out by the Plessey Company represents a major initiative to examine the key issues associated with the transition to Ada. In common with views expressed elsewhere [9], Ada is regarded as much more than another programming language and is expected to provide a new and real opportunity to catalyse substantial improvements in industry's software engineering capability. In particular, by allowing the unification of working practices, Ada is expected to increase the opportunities for much greater levels of software portability and re-use, software reliability, and overall productivity.

Industry is keen to take advantage of these benefits but clearly any significant advance requires the availability of (cross-) compilers and support tools appropriate to the systems to be produced. Such compilers need to be not only technically compliant with the Ada Language Reference Manual but also of high performance and operationally efficient.

This is still an area of concern and despite the (increasing) number of validated Ada compilers available it is thought that there is much to be done before Ada can really be put to effective use for embedded microprocessor applicat-ions for which Ada has most to offer.

However, for the complex and real-time applications described here, the experience gained to-date gives cause for optimism. Thus, in the development of the Vax-based platforms the DEC Ada compiler appeared to be well engineered and operationally efficient. In these particular circumstances it resulted in comparable run-time performance in at least one application to Pascal, and although it currently involves increased code sizes when the run-time allocation of storage is taken into account, it is thought likely that this situation will improve in future products.

At the same time, the fact that the research programme has slipped in time-scales due to the non-availability of high performance cross-compilers and tools to support representative embedded micro-processor configurations is a cause of concern. Such delays could frustrate industry in bidding for and implementing these types of application. A further concern is that there often seems to be insufficient documentation concerning the detailed compiler characteristics eg resource requirements, availability of intermediate compiler outputs, run-time support features etc. This is an area which is just beginning to receive greater attention in the Ada community and needs to be encouraged.

As a general conclusion, it is felt that to-date there has been considerable emphasis (perhaps not surprisingly) on producing validated Ada compilers, but much less on providing genuine support for real Ada developments (whether this be appropriate compilers, support tools or in devising effective working methods for use with Ada). Clearly, if Ada is to be put into real service and provide the benefits that industry is expecting this balance has to be redressed at the earliest opportunity.

Acknowledgements

This paper is produced as a consequence of an in-house (private venture) funded Ada research programme carried out by the Plessey Company in the U.K. The author would like to record his thanks to the Company for supporting the production and presentation of this paper, and to those people engaged on the programme who have provided the results presented.

References

1. Official Definition of Mascot, The Mascot Suppliers Association, U.K., 1980.
2. Cooper G A, Carter C B, Hess A, "AV-8B/GR Mk 5 Engine Monitoring System", 21st Joint Propulsion Conference, California, 1985.
3. Slape J, "Experience in the Use of Ada for a Digital Switching System", Ada UK Conference, 1986.
4. Yourdon E and Constantine L, Structured Design, Prentice-Hall, 1979.
5. DeMarco T, Structured Analysis and System Specification, Prentice-Hall, 1979.
6. Learmonth and Burchett's Structured Design Method (LSDM), Learmonth and Burchett Management Systems, London.
7. Booch G, Software Engineering with Ada, Benjamin/Cummings, California, 1983.
8. Berard E V, An Object-Oriented Design Handbook for Ada Software, E.V.B. Software Engineering Inc., 1985.
9. Walsh T J, "Transitioning to Ada: The challenge for Software Engineering", Proc. 3rd Annual National Conference on Ada Technology, Texas, 1985.

ADA® IN USE; A DIGITAL FLIGHT CONTROL SYSTEM

T. F. Westermeier and H. E. Hansen
McDonnell Aircraft Company
McDonnell Douglas Corporation
P.O. Box 516
St. Louis, MO 63166

SUMMARY

A microprocessor-based, parallel-processing flight control system has been built around the F-15 Eagle Dual Control Augmentation System and has been successfully flight tested. The microprocessors are programmed using Ada, the Department of Defense (DoD) standard high order language. It is widely agreed that Ada has the potential for reducing software life cycle costs through increased programmer productivity. To use Ada and realize the productivity gains, however, the compiler must be reasonably efficient. The use of Ada is discussed, therefore, from these two interrelated standpoints: software productivity and compiler efficiency. The productivity gains and the level of efficiency actually achieved are highlighted.

INTRODUCTION

The first flight of the U.S. Air Force/McDonnell Douglas F-15 with an Ada-programmed flight control system occurred at Edwards Air Force Base, California, on September 18, 1984. The F-15 thus became the first aircraft to fly with a mission-critical system programmed in the Ada computer language.

The compiler and supporting equipment used in the flight were supplied by Zilog, Inc. The compiler was designed and built by Irvine Compiler Corporation. At the time of this writing certification of the Ada compiler is in process and validation is expected to be completed in 1986. The operational flight software was developed by McDonnell Aircraft, and is used in the flight control system's four Zilog Z8002 microprocessors. The hardware used in the system was developed by the Astronics Division of Lear-Siegler, Inc. The same system, programmed in the Pascal computer language, flew for the first time in May 1983.

Recognizing that a number of technological advancements were occurring which could importantly affect the direction of digital flight control system design in the near future, the McDonnell Aircraft Company initiated an Independent Research and Development (IRAD) program in the summer of 1981 designed to assess advancements in the following technology areas: microprocessors, higher order languages (HOL's), floating point arithmetic, and parallel processing.

It has long been recognized that HOL's have the potential for reducing software life cycle costs, and current language standardization thrusts within the DoD are in pursuit of that goal. On the other hand, the inefficiency of compilers has often meant that HOL's were not feasible or cost effective for embedded software applications. Because of recent developments, including higher quality compilers, cheaper memory, higher speed processors, and computer architectures which better support HOL's, a reassessment of the use of HOL's was undertaken as an important part of the IRAD activity.

In conducting this reassessment, the requirement to use Ada in future defense systems was kept strongly in mind. The DoD has mandated the use of Ada in all mission-critical defense systems that enter advanced development status after January 1, 1984 or that enter full-scale engineering development status after July 1, 1984.

At the time of initiation of the IRAD, suitable Ada compilers were not available. Consequently it was decided to begin the HOL assessment with Pascal since, from a user's standpoint, Pascal has many of the features of Ada. The particular Pascal compiler used was a 7-pass, optimizing, microconcurrent version, built by Enertec, Inc., Lansdale, PA. An Operational Flight Program (OFP) (including the executive, control laws, and built-in-test) was written in Pascal and flown on the F-15 Eagle.

In late 1983 an Ada compiler was procured from Irvine Compiler Corporation, Irvine, California. Compiler efficiency was made the first priority. Whatever the virtues of a high-order language, the use of a HOL (including Ada) becomes impractical in many real-time embedded applications if the compiler is inefficient, i.e., if the object code produced by the compiler occupies too much memory or requires too much computation time. In particular, a digital flight control system in many ways supplies the most severe test to the use of Ada in real-time applications.

A complete OFP, including an executive, control laws, and a built-in-test function was written in Ada. The object code was first tested at the module level. The software was then integrated with the hardware and the combination subjected to a series of tests befitting a flight-critical system, including a closed-loop, man-in-the-loop simulation. The system was then ground tested on the airplane prior to flight testing.

®Ada is a Registered Trademark of the U.S. Government (Ada Joint Program Office).

The purpose of the present paper is to report our findings as to the use of Ada in a Digital Flight Control System (DFCS). The specific hardware configuration that served as the supporting system for Ada is described first. The configuration is shown to be a microprocessor based, parallel processing system that facilitates an efficient software structure. The bulk of the paper then concentrates on the evaluation of Ada from two interrelated standpoints: software productivity and Ada efficiency. The productivity gains and the level of efficiency actually achieved are highlighted.

Because of the stringent hardware size and computation time requirements inherent in DFCS's, these systems in many ways supply the litmus test to the use of Ada in other embedded real-time applications.

Implementation of Ada in the flight control system is the first step towards an Ada-based integrated control system that will include elements of fire control, aided navigation, propulsion and trajectory control.

SYSTEM DESCRIPTION

Hardware Configuration - The particular hardware configuration that served as the supporting system for technology evaluation is built around the present F-15 Eagle dual Command Augmentation System (CAS). This was done for two reasons: the electronic portion of the CAS could be modified at minimum cost/time to provide all of the required research and development digital features; and, the system could be flight tested with minimum aircraft modification.

The present production F-15 control system is shown in Figure 1. The CAS computers have been modified as depicted in Figure 2. The existing dual analog computers in each computer LRU were replaced with two digital processors (Z8002 microprocessors) (Reference (1)) and associated memory and converters. The modifications were carried out by the Astronics Division of Lear Siegler Corporation, Santa Monica, and are described in detail in Reference (2).

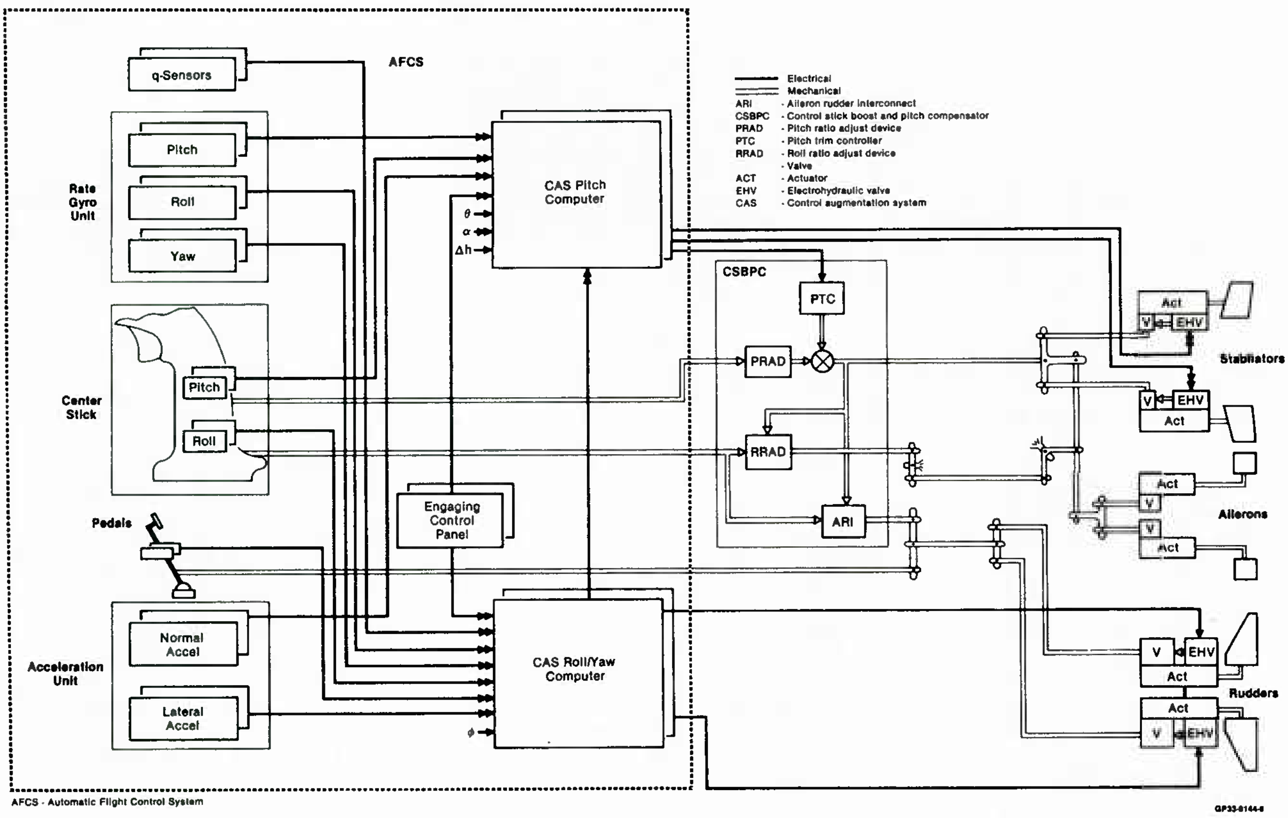


Figure 1. F-15 Present Mechanical and Dual Command Augmentation Flight Control System

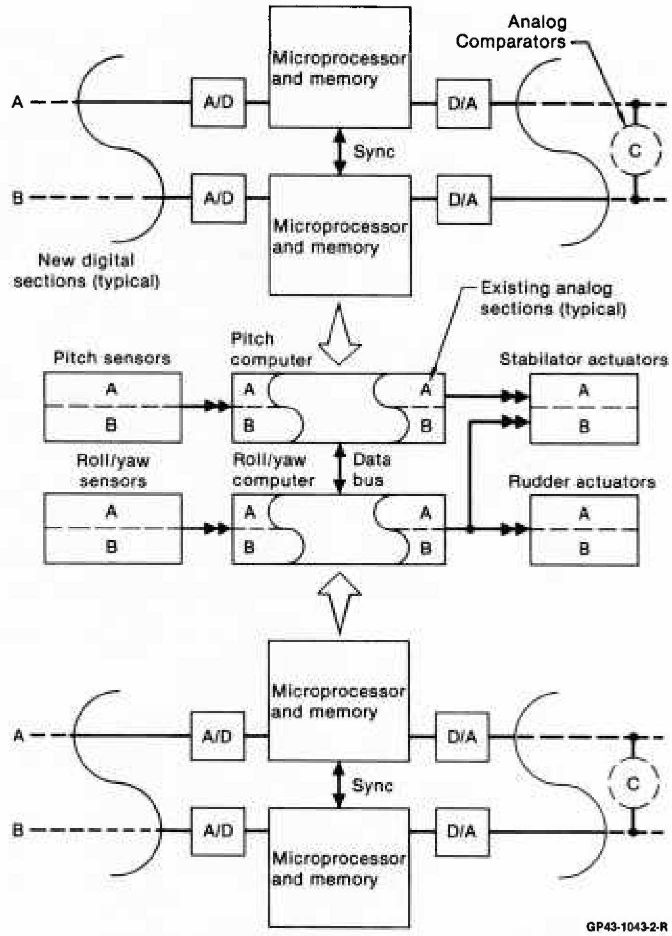


Figure 2. F-15 Flight Control Computer Modifications

The resulting system is a parallel processing system with the Operational Flight Program (OFP) partitioned by control axes[3]. Two processors operate on the pitch axis software and two operate concurrently on the roll/yaw software. The two pitch processors are frame synchronized, as are the roll/yaw processors. The two pitch processors, however, are unsynchronized with respect to the roll/yaw processors. The dual processors in the pitch and roll/yaw axes are consistent with the dual (fail-safe) nature of the F-15 production CAS.

Reference (3) shows that the increased computation power afforded by parallel processors (vis-a-vis a uniprocessor) can be used to advantage in a number of ways. The increased computation power afforded by the present parallel system has been exploited to support an HOL and to simplify the software structure. For example, all control laws are processed at a single iteration rate, thereby avoiding a multiple rate structure.

<u>Operational Flight Program</u> - The OFP includes an executive, control laws, and a built-in-test (BIT) function all programmed in Ada. The control laws used in this Ada version of the OFP are designed to emulate the analog control laws, and therefore the flying qualities of the production F-15. The built-in-test function includes a pre-flight and a maintenance BIT, as described in detail in Reference (4).

The computational events in a 12.5 ms frame are depicted in Figure 3. All parts of the control laws are computed at 80 Hz.

Memory sizes and computation times for the Ada version of the OFP are given in Figure 4. (The maintenance portion of BIT is not included in this figure). Obviously, a large amount of spare memory is available for future growth. The computation time requires a 38% and 59% duty cycle in the pitch and roll/yaw computers. Thus, adequate spare computation time is also available for future growth. The 622 word kernel referred to in Note 4 is primarily the floating point algorithm written in assembly language.

FLIGHT TEST RESULTS

The DFCS programmed with the Ada OFP of Figure 4 was flight tested at Edwards AFB during 1984. As shown in Figure 5, nine dedicated DFCS/ Ada test fights were flown in two phases. The DFCS/Ada configuration was flown on nine other missions whose purpose was other than the specific evaluation of DFCS/Ada.

The first test phase included four DFCS/Ada flights. The maneuvers included middle of the envelope handling quality checks up to 1.2 Mach, large amplitude maneuvering, tracking and formation flight. In all cases the test pilots commented that the aircraft flew the same as any other F-15 with an analog flight control system.

Following the initial four DFCS/Ada flights it was decided to fly some additional flights in order to expand the flight envelope. The second phase of DFCS/Ada testing consisted of five flights.

Two flights (668-7 and 671-8) were flown in order to investigate the high angle-of-attack handling qualities of the system with a clean aircraft configuration and a configuration with a centerline tank and two inboard pylons. The pilots indicated that they could not tell the difference between an aircraft equipped with an analog or digital CAS.

The final test flight (672-9) in this series of DFCS/Ada test flights expanded the DFCS flight envelope to the high supersonic speeds. Five speed runs along with three air refuelings were performed on this flight. The DFCS equipped aircraft was flown to calibrated airspeeds up to 720 knots, Mach numbers up to 2.0 and at altitudes exceeding 45,000 feet. At all test points there were no anomalies noted in the data and pilot comments indicated that the aircraft response was the same as a standard F-15 with an analog CAS.

All the test points flown in these last four instrumented flights are plotted on a standard F-15 level flight envelope in Figure 6. At all test points the AFFTC test pilots noted that the DFCS aircraft responses were the same as a standard F-15. These excellent results indicate that the DFCS programmed in Ada can be utilized throughout the F-15 flight envelope.

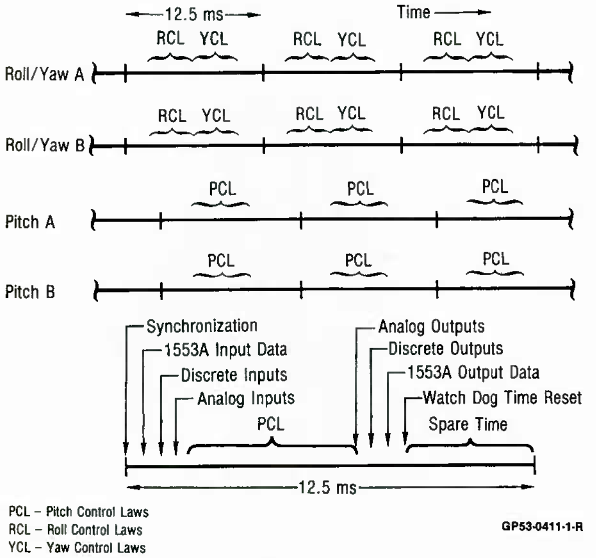


Figure 3. Events in 80 Hz Computation Frame

	OFP Element	Size (16 Bit Words)	Time (ms)
Pitch Computers (Each)	Pitch CAS Control Laws	1,343	2.81
	Executive	2,026	2.00
	Built-In-Test (Preflight)	934	N/A
	Total	4,925(4)	4.81
	Available	27,648(2)	12.50(3)
	Spare	22,723	7.69
Roll/Yaw Computers (Each)	Roll CAS Control Laws	775	1.72
	Yaw CAS Control Laws	867	2.66
	Executive	2,454	3.02
	Built-In-Test (Preflight)	995	N/A
	Total	5,713(4)	7.40
	Available	27,648(2)	12.50(3)
	Spare	21,935	5.10

Notes:
(1) OFP version Ada; uses MCAIR floating point algorithm
(2) 27K flight memory card: 24K PROM; 2K scratch pad; 1K NVR
(3) 80 Hz computation frame
(4) Includes 622 word kernel

GP43-1043-4-R

Figure 4. DFCS OFP(1) Memory Size and Computation Time

Date	Flight Number	Flight Time	Pilot	Purpose
9/17/84	622-1	1.2	Lt Col Saxon	Middle of Envelope to Low Supersonic Testing
9/18/84	623-2	1.1	Flt Lt Sears	Middle of Envelope to Low Supersonic Testing
9/19/84	624-3	1.0	Flt Lt Sears	Tracking and Formation
9/19/84	625-4	0.7	Lt Col Saxon	Large Amplitude Maneuvering
11/19/84	659	1.6	Maj Strittmatter	Non-Test Flight
11/20/84	660	1.7	Maj Ferraioli	Non-Test Flight
12/3/84	661	0.4	Maj Strittmatter	Non-Test Flight
12/4/84	662	2.6	Maj Strittmatter	Non-Test Flight
12/4/84	663	0.4	Maj Strittmatter	Non-Test Flight
12/4/84	664-5	1.2	Maj Ferraioli	Middle of Envelope to Low Supersonic Testing
12/5/84	665-6	1.5	Flt Lt Sears	Tracking and Formation
12/5/84	666	0.4	Maj Strittmatter	Non-Test Flight
12/6/84	667	0.4	Maj Strittmatter	Non-Test Flight
12/12/84	668-7	1.4	Flt Lt Sears	High Angle-of-Attack
12/12/84	669	1.6	Maj Prill	Non-Test Flight
12/17/84	670	1.1	Flt Lt Sears	Non-Test Flight
12/17/84	671-8	1.8	Maj Strittmatter	High Angle-of-Attack
12/20/84	672-9	1.5	Flt Lt Sears	Supersonic
	18 Flights	21.6 hr	5 Pilots	

GP43-1043-26-R

Figure 5. Summary of DFCS/Ada Flight Testing

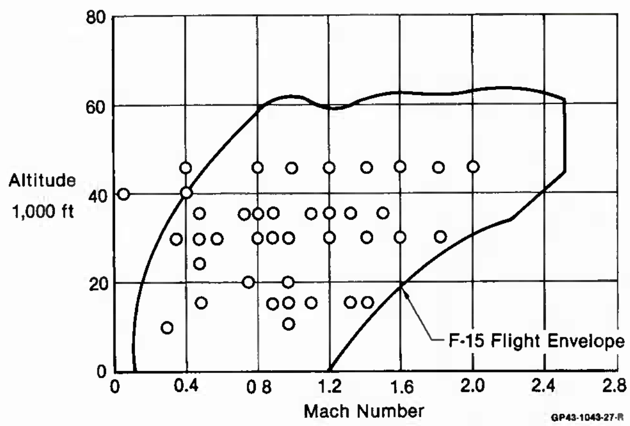


Figure 6. DFCS/Ada Flight Test Condition

SOFTWARE PRODUCTIVITY

An increase in software productivity and a corresponding decrease in the cost of developing software is, of course, the raison d'etre for the use of Ada. Studies made for the DoD predict that Ada will result in substantial cost savings DoD-wide.

A number of analyses have been made of the impact of HOL'S on software productivity and life-cycle costs. Software productivity analyses necessarily involve assumptions which are often highly judgemental, and sweeping generalizations about software life-cycle costs are rightly viewed with skepticism.

With these caveats in mind, we present in Figure 7 our experience of the impact of Ada on programmer productivity. The DFCS program is admittedly not a large experience base and certainly no claim is made that these results are universally true. The figure says that a software program written in floating point assembly language or Pascal will require only about 40% and 20% of the effort required to produce that same program in fixed point assembly language. Because of the similarity of Ada and Pascal from the user's standpoint, for a flight control application Ada productivity gains are about the same as Pascal. Assembly language is used as the basis of comparison since it is used in the vast majority of embedded applications.

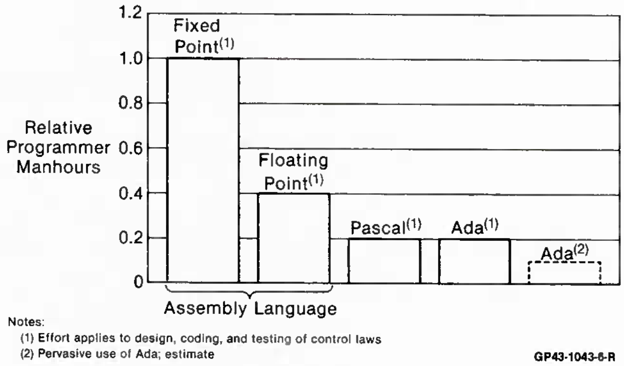


Figure 7. Impact of HOL and Floating Point on Programmer Productivity

For DFCS applications, the two features of Ada that strongly impact productivity are high level constructs and floating point. As illustrated in Figure 8, they eliminate the labor associated with fixed point arithmetic and simplify and reduce the source statements required to program a system, thereby reducing programming errors and simplifying program documentation. In (a), a structural filter is shown in its original s-plane form; (b) shows the difference equation representation that is solved in the digital computer; (c) shows the single Ada program statement required to solve this equation; (d) shows the number of assembly language statements required. Note that the assembly language employs floating point add/subtract (FADD/FSUB) and floating point multiply (FMUL) instructions. If fixed point arithmetic had been used, the number of statements required would have been still greater.

The Ada language is designed to promote the goals of software portability and reuseability. When these goals are pervasively realized, it is estimated by a number of notable software engineers that Ada may achieve productivity gains of 10:1 vis-a-vis assembly language. Currently, however, a major impediment to the realization of these goals is the lack of a large number of high quality compilers and associated target computers.

<u>The Phases of Software Development</u> - It is widely agreed that the total of software development consists of four activity phases: design, coding, testing and maintenance. Ada obviously has a large impact on coding productivity as illustrated in Figure 8. Less obvious is the influence on software design, testing and maintenance. The impact on software productivity must therefore be assessed for each of the four phases of software development. The assessment must also take into account the four functional divisions of the total flight control OFP: the control laws, the executive, the redundancy management function, and the built-in-test (BIT) function.

Original S-Plane Representation of Structural Filter

$$\frac{0.4807S^2 + 83.5533S + 3894}{S^2 + 125S + 3894}$$

Difference Equation Representation

STFL = 0.56503 • PRESTRU
− 0.33991 • PREM1STRU
+ 0.089533 • PREM2STRU
+ 0.87711 • STFM1 − 0.19182 • STFLM2

or

Ada Statement

STFL: = 0.56503 • PRESTRU
− 0.33991 • PREM1STRU
+ 0.089533 • PREM2STRU
+ 0.877116 • STF1M1 − 0.19182 • STF1M2;

Assembly Language Statements*

STFL = [0.56503 • PRESTRU] − [0.33991 • PREM1STRU] + [0.089533 • PREM2STRU] + [0.87711 • STFLM1] − [0.19182 • STFLM2]

```
LDL   RR10,PCAS24;      % RR8 Contains PRESTRU
CALL  FMUL;             % 0.56503 • PRESTRU
LDL   RR6, RR8;         % Store Result for Later Use
LDL   RR8, PREM1STRU;
LDL   RR10, PCAS25;
CALL  FMUl;             % 0.33991 • PREM1STRU
LDL   RR10,RR8;         % Prepare for Subtraction
LDL   RR8,RR6;
CALL  FSUB,             % RR6-[0.33991 • PREM1STRU]
LDL   RR6,RR8;          % Store Result for Later Use
LDL   RR8,PREM2STRU;
LDL   RR10,PCAS26;
CALL  FMUL;             % 0.089533 • PREM2STRU
LDL   RR10,RR6,
CALL  FADD              % RR† + [0.089533 • PREM2STRU]
LDL   RR6,RR8;          % Store Result for Later Use
LDL   RR8,STFLM1;
LDL   RR10,PCAS27;
CALL  FMUl;             % 0.87711 • STFLM1
LDL   RR10,RR6;
CALL  FADD;             % RR6 + [0.87711 • STFLM1]
LDL   RR6,RR8           % Store Result for Later Use
LDL   RR8,STFLM2;
LDL   RR10,PCAS28;
CALL  FMUl;             % 0.19182 • STFLM2
LDL   RR10,RR8;         % Prepare for Subtraction
LDL   RR8,RR6;
CALL  FSUB;             % RR6 − [0.19182 • STFLM2]
LDL   STFl,RR8;
```

*Based on floating point algorithm
*Twice as long if fixed point

GP43-0013-25

Figure 8. Ease of Programming; Ada vs Assembly Steps in Digital Mechanization of Structural Filter (STFL)

A qualitative assessment is made in Figure 9 of the impact of Ada on each of the functional divisions in each of the development phases. The percentages shown for each of the functions are typical amounts, by word count, of the total OFP; thus, the control laws are typically 20% of the total OFP size. The percentages shown for the development phases are estimates of the amount of the total software development activity. Note that the maintenance phase is less of a percentage of the total job than it might be for many types of embedded software applications. This is so since the DFCS maintenance phase is considered to begin with delivery of the first production aircraft. Barring major structural, equipment, or aerodynamic changes to the aircraft, changes to flight control software should be minor for production aircraft. Nevertheless, when these changes are required, they are much easier to make with Ada.

OFP Functional Divisions		Software Development Phases			
		Design (25%)	Coding (20%)	Test(1) (45%)	Maintenance(2) (10%)
Control Laws	(20%)	Some	Large	Large	Some
Executive	(15%)	Nil	Large	Some	Nil
Redundancy Management	(20%)	Nil	Large	Some	Nil
Built-in Test	(45%)	Nil	Large	Some	Some

Notes:
(1) The test phase includes development flight testing
(2) The maintenance phase begins with the delivery of production aircraft

GP63-0004-1

Figure 9. A Qualitative Assessment of the Impact of Ada on Software Productivity

Ada and floating point also have some impact on software design, testing and maintenance, although the impact is less direct. The language used has negligible influence on the design of executive, redundancy management, and BIT software. Synthesizing the control laws from aerodynamic data is totally independent of language considerations; however, rendering the usual s-plane representation of the control laws into a digital format is made easier with Ada and floating point.

Some aspects of testing and maintenance are made easier through the use of Ada and floating point. During testing of the control laws, for example, finding and eliminating overflows occasioned by the use of fixed point can be very time consuming. Modifications to the software are much easier to make with Ada and floating point since only the affected area of the program need be changed, whereas with assembly language and fixed point, substantially more of the program may have to be rescaled and reprogrammed.

Concern is often expressed that testing, especially at the systems level, is more difficult if Ada is used; an error found during hardware-software integration, for instance, may be harder to locate and correct in the object code. This problem can be circumvented by having the compiler generate an assembly-language listing that includes comments that link the assembly-language object statements to the Ada source statements, as in Figure 8(d). Such listings are necessary in any case if patches to the program are to be made. In the DFCS project, no testing difficulties were encountered because of Ada.

Self Documentation - If properly generated, an Ada PDL Specification contains the information and has the clarity of a flow chart. The compiler generates the listing and programmer labor is not required. In addition, the source code by its very nature always reflects the underlying machine instructions. The source statements in the listing, moreover, are readily traceable back to the original software design document.

Transportability - An OFP written in assembly language must be checked out on the actual target processor, an emulator of the exact processor, or a software simulator of that processor. In all cases, the source program cannot be tested until one or more of the above are procured and this may cause a delay in software development. With Ada the problem can be circumvented. If a compiler is available for any target machine, simulator, or emulator, the source program can be tested using one of these devices. In so doing, a high confidence exists that the source program is correct well before the target machine becomes available.

Floating Point Arithmetic - Floating point arithmetic is supported by most HOL's, including the Ada language. In addition, floating point offers a number of advantages vis-a-vis fixed point, even when used with assembly language: the range of numbers that can be handled is extended; inherent safety features are provided; and software productivity is increased.

The use of floating point arithmetic extends the range of the variables used in DFCS software with some important consequences: as examples, the granularity of a variable is decreased, and the significant figures in a variable are more easily preserved.

The use of floating point algorithms also provides inherent safety features, particularly the handling of overflows. Overflows, for example, are prevented from occurring in the body of the program.

Perhaps the biggest advantage of floating point, however, is increased programmer productivity. With floating point, scaling is automatic, and the programmer is relieved of all tedious and time consuming scaling considerations. This is in keeping with the philosophy of having the computer handle all the tasks of which it is capable, allowing the engineer more time for productive tasks.

It is a common perception that fixed point arithmetic is always faster than floating point. This is not necessarily true. In fact, although floating-point addition and subtraction may take longer than fixed-point, floating-point multiplication and division may be as fast or faster than fixed-point. In addition, fixed-point programs often employ time-consuming double precision arithmetic to minimize round-off and truncation errors. This is not to suggest, of course, that the speed of the floating point algorithm used is unimportant.

It was always the intent on the DFCS program to eventually implement floating point in hardware; however, at the time of initiation of the program, suitable hardware was not available. In the interim, various software floating point algorithms have been used.

The first floating point algorithm to be used employed the IEEE format and a 24-bit mantissa. It was subsequently modified to use a 16-bit mantissa. This modified algorithm achieved faster speeds with little degradation in accuracy and for a time was the baseline implementation.

Recognizing that the speed of a floating point algorithm could be improved by tailoring the format to the computer architecture, Harold Hansen, one of the authors of this paper, developed a floating point algorithm which dramatically reduces computation time from that achieved with the 16-bit algorithm, and which even approaches the performance expected from a fast co-processor (see References 5 and 6). The mechanization has been used in the Ada, Pascal, and assembly language versions of the OFP with very favorable results.

The Ada language supports the use of fixed point arithmetic; however, the authors believe that the use of fixed point, except in isolated instances, represents a major step backwards. Therefore, the potential user of Ada should give serious consideration to how floating point is to be mechanized.

COMPILER EFFICIENCY

Efficiency Goals - The main disadvantage of HOL's compared to assembly language is an expansion of program size and execution time. The magnitude of expansion for a given HOL varies widely, depending on the application, the compiler, and the target computer. In DFCS applications, program size directly affects recurring costs. At some point, the additional cost of memory needed to accommodate a HOL for some number of aircraft can offset the software development cost savings afforded by a HOL. If DFCS performance is not to be compromised, the expansion of execution time must be held to a tolerable level. Memory and time expansion goals were therefore established, below which HOL code would be cost and performance competitive with assembly language code, to wit:

GOALS:

-- Time Expansion < 20%
-- Memory Expansion < 50%

The goals are about what we think the efficiency of a compiler should be in order to produce object code that is competitive with assembly-language code on a production DFCS. The rationale used to establish these goals follows.

In DFCS applications, computation time is of the essence. For reasons of minimizing the deleterious effects of aliasing and transport lags, DFCS computers must operate at fairly high computation (iteration) rates (80 iterations per second are common). All foreground portions (control laws, executive, and redundancy management) of the OFP must be completed within the computation frame.

It is, of course, difficult to generalize about how much time expansion can be tolerated. In the present DFCS, an expansion in time in excess of 50% could be tolerated. On the other hand, systems that just satisfy MIL-F-9490D would run out of time if the expansion were in excess of 25%. For the purpose of the present study, an expansion goal of less than 20% (expansion factor 1.2) was established.

As stated earlier, the most direct and measurable effect of the expansion in program size is the increased recurring cost of memory. When contemplating the use of a HOL, the trade-off between reduced software development costs and increased recurring costs must be made, and the point determined at which a HOL becomes cost-effective. This point will, of course, differ for every system application.

The factors involved are illustrated with Figure 10. The important assumptions involved are shown on the figure. For simplicity, the graph is made linear, thereby neglecting the fact that at an expansion factor of 2.0, say, a whole new PROM board might be required per computer; whereas at lower expansion factors, 1.1 say, only additional memory chips may be required to be added to an existing board. In this illustration, if, for example, the expansion factor is 2.0 (100%) the recurring cost of additional memory is $2.4 MIL for a quad system. It is doubtful if the use of a HOL would reduce the development cost of an OFP of this size so as to offset the increased memory costs. On the other hand, with an expansion factor of 1.5 or less, the HOL might pay for itself. Based on this type of analysis, a memory expansion goal of less than 50% was established.

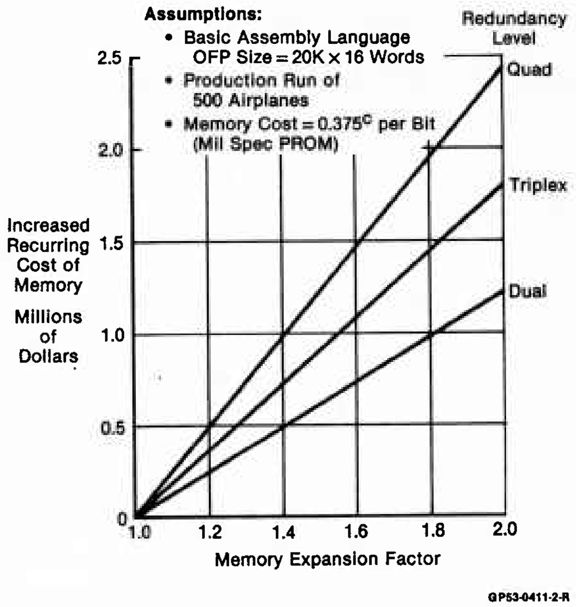


Figure 10. The Impact of HOL Inefficiency on the Increased Recurring Cost of Memory

Efficiency Achieved - The computation time and memory expansion numbers for the Ada OFP that was flown (Figure 4) are 1.1 and 1.63 as shown in Figure 11. The time expansion is seen to be below the goal. The memory expansion is currently above the goal. In the expansions, the assembly language program used for comparison employs floating point.

HOL Expansions Compared to Floating Point Assembly Language

		Pascal			Ada
	Goal	February 1983	May 1983	October 1983	August 1984
Memory Expansion	<1.5	1.77	1.35	1.30	1.63 (1.36)*
Time Expansion	<1.2	1.28	1.16	1.10	1.10

*Compared to fixed point assembly language

GP43-1043-5-R

Figure 11. Compiler Efficiency

Achieving a high level of Ada efficiency will be an ongoing activity as it was with Pascal, with similar results expected. Figure 11 illustrates how Pascal expansions decreased over time as the compiler was modified for improved efficiency. As shown, by Oct 83 the memory and time expansions had been reduced to 1.30 and 1.10, well below the established goals. Ada time expansion is already below the goal.

SUMMARY AND CONCLUSIONS

An operational flight program has been developed in Ada and flown on an F-15 Eagle, in evidence of the fact that Ada is feasible in DFCS applications and therefore in other avionic systems as well. In so doing, the increased software productivity resulting from the use of Ada has been demonstrated.

In general, DFCS applications impose stringent requirements on compiler efficiency. If memory expansion is too large, the increased recurring cost of system hardware will offset the productivity gains; if computation time expansion is too large, the performance of the system will suffer. In the present instance, exemplary memory and computation time expansion levels have been achieved with an efficient Ada compiler.

The authors are of the opinion that the use of Ada holds great promise in avionics systems. At the same time, they realize that Ada is not a panacea. If a computation task cannot be accomplished in assembly language, neither can it be accomplished in Ada. If the compiler is inefficient, or if the target computer speed is inadequate, then the benefits of Ada may be elusive. But under the right combination of computer speed, efficient software structure, and compiler efficiency, as in the present F-15 DFCS, the benefits of Ada are substantial and can be realized.

REFERENCES

(1) Advanced Micro Devices, "AM Z8000 User's Manual", AMD Publication Number AMZ-291, April 1981.

(2) Lear Siegler, Inc. (Astronics Division), "F-15 DEFCS Hardware and Interface Description", Report No. ADR-843/1, dated 27 July 1982.

(3) Westermeier, T.F., "Parallel Processing Applied to Digital Flight Control Systems; Some Perspectives", NAECON, May 1981.

(4) Hansen, H. E., "A Method For Testing A Digital Flight Control System Without The Use of Ground Support Equipment", IEEE/AIAA 6th Digital Avionics System Conference, November 1984.

(5) T. F. Westermeier, H. E. Hansen, "Recent Digital Technology Advancements and Their Impact on Digital Flight Control Design", NAECON, May 1983.

(6) T. F. Westermeier, H. E. Hansen, "The Use of High Order Languages In Digital Flight Control Systems", IEEE/AIAA 5th Digital Avionics System Conference, November 1983.

SELECTIVE BIBLIOGRAPHY

This bibliography with Abstracts has been prepared to support AGARD Lecture Series No. 146 by the Scientific and Technical Information Branch of the US National Aeronautics and Space Administration, Washington, D.C., in consultation with the Lecture Series Director, Theodore F. Westermeier, McDonnell Aircraft Company, St. Louis, Missouri.

PRINT 03/4/1-82 TERMINAL=67

UTTL: Reference manual for the Ada programming language CORP: Honeywell Systems and Research Center, Minneapolis, Minn.; Alsys, Saint Cloud (France).

ABS: Ada is a programming language designed in accordance with requirements defined by the United States Department of Defense: the so-called Steelman requirements. Overall, these requirements call for a language with considerable expressive power covering a wide application domain. As a result, the language includes facilities offered by classical languages such as Pascal as well as facilities often found only in specialized languages. Thus the language is a modern algorithmic language with the usual control structures, and with the ability to define types and subprograms. It also serves the need for modularity, whereby data, types, and subprograms can be packaged. It treats modularity in the physical sense as well, with a facility to support separate compilation. In addition to these aspects, the language covers real-time programming, with facilities to model parallel tasks and to handle exceptions. It also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, both application-level and machine-level input-output are defined.

RPT#: AD-A131511 83/01/00 84N10800

UTTL: Proceedings of the 2nd Annual Conference on Ada (Trademark) Technology CORP: Army Communications-Electronics Command, Fort Monmouth, N.J. CSS: (Center for Tactical Computer Systems.) Conf. held in Hampton, 27-28 Mar. 1984

RPT#: AD-A142403 84/03/00 84N30745

UTTL: Ada (registered trademark) technical overview. L102 teacher's guide CORP: Softech, Inc., Waltham, Mass.

ABS: An introduction to the Ada computer programming language is given. A summary of Ada program features is provided. The Ada and FORTRAN languages were compared.

RPT#: AD-A141862 84/05/00 84N29532

UTTL: Introduction to Ada, a higher order language. L103 teacher's guide CORP: Softech, Inc., Waltham, Mass.

ABS: This guide aims to assist instructors who present courses in the Ada programming language.

RPT#: AD-A141848 84/05/00 84N29531

UTTL: Ada (Trademark) case studies II CORP: Softech, Inc., Waltham, Mass.

ABS: This report presents a set of case studies on different aspects of the Ada language. The work which led to this volume was performed by SofTech, Inc. under the Ada Design Methods Training Support Contract. The objective of this report is threefold: To expand on Ada issues which surfaced in the ASDMF contract cited above to explore new Ada issues to gain insight into the characteristics of a life cycle design methodology that would promote effective use of Ada this report contains fifteen case studies which illustrate different areas of the Ada language: naming conventions; types; coding paradigms; exceptions; and program structure.

RPT#: AD-A140818 84/01/00 84N27475

UTTL: Ada compiler validation summary report: ROLM Ada compiler, version 4.52 V-003 CORP: Softech, Inc., Waltham, Mass.

ABS: The ROLM Ada compiler, version 4.52, was tested in June 1983 with version 1.1 (March 4, 1983) of the ACVC validation tests. Version 1.1 of the test suite contained 1,595 tests, of which 1,292 were applicable to this compiler. Of the applicable tests, 56 were withdrawn due to errors in the tests. All of the remaining 1,235 applicable correct tests were passed.

RPT#: AD-A136760 83/06/03 84N20202

UTTL: Ada compiler validation report: Western digital STC-Ada compiler, version C1.0M V-004 CORP: Softech, Inc., Waltham, Mass.

ABS: This report describes the results of the validation effort for the following Ada translator: Host Machine: Western Digital WD1600 Series MicroEngine; Operating System: STC Ada Operating System 2.9; Host Disk System: 10 megabyte Winchester; Target Machine: Western Digital WD1600 Series MicroEngine; Operating System: STC Ada Operating System 2.9; Language Version: ANSI/MIL-STD-1815A Ada; Translator Name: STC-Ada; Translator Version: C1.0m; and Validator Version: 1.1 (March 4, 1983). Testing of this translator was conducted by SofTech, Inc. The purpose of this report is to document the results of the

testing performed on the translator, and in particular, to: identify any language constructs supported by the translator that do not conform to the Ada standard; identify any unsupported language constructs required by the Ada standard; and describe implementation-dependent behavior allowed by the standard.
RPT#: AD-A136738 83/07/28 84N20200

UTTL: Computer program development specification for Ada integrated environment: Program integration facilities, type b5, B5-AIE (1). PIF (1) CORP: Intermetrics, Inc., Cambridge, Mass.
ABS: This document establishes the performance, design, test, and qualification requirements for the Program Integration Facilities for the Ada Integrated Environment. These facilities include the Program Library Interface Packages, the Program Builder and the Program Library Support Tools. This document also includes requirements for the design of the Ada program library, the environment within which program integration occurs.
RPT#: AD-A134003 IR-681-1 83/03/22 84N14774

UTTL: Computer program development specification for Ada integrated environment. Ada compiler phases B5-AIE (1). COMP (1) CORP: Intermetrics, Inc., Cambridge, Mass.
ABS: This document specifies the requirements for the performance and verification of the Ada compilers for the IBM (VM/370) and Perkin-Elmer (PE) 8/32 (OS/32) systems. Each compiler provides the user with the ability to translate an Ada compilation and obtain a program listing and linkable machine code for the respective target machine; listing, optimization, and debugging control are selectable by the user. Because of the compiler structure and the similarity of the target machines, the two compilers are nearly identical. As a result, this document presents the design as though there were a single Ada compiler; where target-machine dependencies make the compilers different, this is pointed out in the discussion.
RPT#: AD-A134032 IR-677-2 82/11/05 84N14773

UTTL: System specification for Ada integrated environment type A AIE(1) CORP: Intermetrics, Inc., Cambridge, Mass.
ABS: This specification establishes the performance, design, development and test requirements for the Ada Integrated Environment (AIE), an integrated set of software tools designed to support the development and maintenance of software written in the Ada Programming Language.
RPT#: AD-A134080 IR-676-2 82/11/12 84N14769

UTTL: ADA (trademark) training curriculum. Programming methodology m203 teacher's guide CORP: Softech, Inc., Waltham, Mass.
ABS: Partial Contents: Review of Software Life Cycle; Coding Phase; Goals; Structured Programming; Control Structures; Coding Style; and Ensuring Reliability.
RPT#: AD-A143581 84/07/00 84N33082

UTTL: Ada compiler validation summary report: TeleSoft (TeleSoft Ada) compiler, version 2.0a for Sun 120 Motorola M68010, using 4.2 BSD UNIX-Sun Version 1.1 CORP: Softech, Inc., Fairborn, Ohio.
ABS: The purpose of this report is to document the results of testing performed on the Ada translator and, in particular, to (1) identify any language constructs supported by the translator that do not adhere to the Ada standard; (2) identify any unsupported language constructs required by the Ada standard; and (3) discuss implementation-dependent behavior allowed by the standard.
RPT#: AD-A146925 84/08/10 85N16484

UTTL: Ada compiler validation summary report, Dansk Datamatik Center, VAX 11 compiler version 1.1 CORP: Softech, Inc., Fairborn, Ohio.
ABS: The purpose of this Validation Summary Report (VSR) is to present the results and conclusions of performing standardized tests of the Dansk Datamatik Center Compiler. The suite of tests known as the Ada Compiler Validation Capability (ACVC), Version 1.4, was used. The ACVC suite of tests is used to validate conformance of the compiler to ANSI/MIL-STD-1815A (Ada). The purpose of the testing is to ensure that the compiler properly implements legal language constructs and that it identify, reject from processing, and label illegal language constructs. The testing also identifies implementation-dependent behavior permitted by the standard. Six classes of tests are used. These tests are designed to perform checks at compile time, during execution, and at link time. The ACVC, Version 1.4, contains 2178 tests, of which 2011 were applicable to this implementation. Of the 2011 applicable tests, 73 were withdrawn due to the occurrence of errors in the tests. Results showed that all of the remaining 1938 valid tests were successfully passed by the Dansk Datamatik Center compiler.

RPT#: AD-A149340 AVF-TAR-04.1184 84/11/06 85N20681

UTTL: Digital Equipment Corporation VAX Ada Compiler Version T0.6-2 VAX-11/785, 11/750, 11/730, using VMS, Version 4.0 and MicroVAX 1, using MicroVMS Version 1.0 CORP: Federal Compiler Testing Center, Falls Church, Va.

ABS: The Digital Equipment Corporation Compiler VAX Ada, version T0.6-2, for the VAX-11/785, 11/780, 11/750, 11/730 using VMS version 4.0 and MicroVAX 1 using MicroVMS version 1.0, was tested with version 1.4 of the ACVC validation tests. Version 1.4 of the test suite contained 2173 tests, of which 2099 were applicable to this implementation. Of the applicable tests, 11 were withdrawn due to errors in the tests. Of the remaining applicable correct tests 2099 passed and no anomaly was discovered.

RPT#: AD-A149375 AD-F300533 TC-84-DEC-390 84/09/12 85N19696

UTTL: Ada (trademark) compiler validation summary report: ALSYS ALSYCOMP-0001 CORP: Bureau d'Orientation de la Normalisation en Informatique, Rocguencourt (France).

ABS: The purpose of this report is to document the results of the testing performed on the compiler, and in particular, to: identify any language constructs supported by the compiler that do not conform to the Ada standard; identify any unsupported language constructs required by the Ada standard; describe implementation-dependent behavior allowed by the standard.

RPT#: AD-A153750 84/12/08 85N29592

UTTL: Ada compiler validation summary report, University of Karlsruhe-GMD/German MOD Siemens - BS 2000 version 7.30 CORP: Industrieanlagen-Betriebsgesellschaft m.b.H., Ottobrunn (West Germany).

ABS: The purpose of this Validation Summary Report (VSR) is to present the results and conclusions of performing standardized tests of the GMD/German MOD Compiler. On-Site testing was performed 27 Sep. 1984 to 8 Oct. 1984 at the GMD Center in Birlinghoven, Germany under the auspices of the Ada Validation Facility (AVF), according to the Ada Validation Office (AVO) policies and procedures. The GMD Compiler (Siemens 7.XXX Version 840404) is hosted on the Center's Siemens 7.571 Computer operating under BS2000 7.1. The suite of tests known as the Ada Compiler Validation Capability (ACVC), Version 1.4, was used. A complete list of tests and results is provided in this report.

RPT#: AD-A153747 84/11/28 85N29591

UTTL: Ada (trademark) compiler validation summary report. Honeywell information systems GCOS6 Ada version 1.1 CORP: Federal Compiler Testing Center, Falls Church, Va.

ABS: The Honeywell Information Systems Compiler GCOS6 Ada version 1.1, for the DPS 6/95, DPS 6/75, DPS 6/45, microSystem 6/20 and microSystem 6/10 using GCOS6 MOD400, was tested with version 1.4 of the ACVC validation tests. Version 1.4 of the test suite contained 2185 tests, of which 2008 were applicable to this implementation. Of the applicable tests, 71 were withdrawn due to errors in the tests. Of the remaining applicable correct tests 1937 passed, and no anomaly was discovered.

RPT#: AD-A153746 OIT/FSTC-84/515 84/12/14 85N29590

UTTL: Ada validation summary report. ALS AdaVAX version 1.1 VAX-11/780 using VAX/VMS release 3.6 CORP: Federal Compiler Testing Center, Falls Church, Va.

ABS: Under U.S. Army Contract DAAK80-C-0507, SofTech, Incorporated has developed an Ada compilation system for the Department of the Army, Headquarters USA, Communications Electronic Command. The compiler, ALS AdaVAX version 1.25, was tested against version 1.4 of the Ada Compiler Validation Capability (ACVC) test suite on a VAX-11/780 operating under VAX/VMS version 3.6. Version 1.4 of the test suite contained 2185 tests, of which 1943 were applicable to this implementation. Of the applicable tests, 74 were withdrawn due to errors in the tests. Of the remaining applicable correct tests, all (1875) passed, and no anomaly was discovered.

RPT#: AD-A153684 OIT/FSTC-84/501 84/12/31 85N29588

UTTL: Ada compiler validation summary report: University of Karlsruhe-GMD/German MOD VAX 11 compiler, version v1.0 CORP: Industrieanlagen-Betriebsgesellschaft m.b.H., Ottobrunn (West Germany).

ABS: The purpose of this Validation Summary Report is to present the results and conclusions of performing standardized tests of the System/German MoD compiler. On-site testing was performed 28 Sep. 84 to 6 Oct. 84 at System KG in Karlsruhe, Germany under the auspices of the Ada Validation Facility (AVF), according to the Ada Validation Office policies and procedures. The System compiler (VAX11 Version 1.0) is hosted on the

Center's VAX-11/750 computer operating under VMS 3.0. The suite of tests known as the Ada Compiler Validation Capability, Version 1.4, was used. The purpose of the testing is to ensure that the complier properly implements legal language constructs and that is identify, reject from processing, and label illegal language constructs. The testing also identifies implementation-dependent behavior permitted by the standard. The AVF concluded that the results obtained show accepted compliance to the February 1983 ANSI Ada Reference Manual.

RPT#: AD-A154370 84/11/12 85N30678

UTTL: Distributed avionics processing using ADA

AUTH: A/ADAMS, S. E.; B/CLAUSING, B. PAA: A/(Intermetrics, Inc., Cambridge, MA) IN: NAECON 1983; Proceedings of the National Aerospace and Electronics Conference, Dayton, OH, May 17-19, 1983. Volume 2 (A84-16526 05-01). New York, Institute of Electrical and Electronics Engineers, 1983, p. 979-983.

ABS: Problems that arise when implementing real-time avionics systems are discussed, with emphasis placed on the issues related to the use of the Ada language for programming single and multiple processor systems. Consideration is given to systematic and random timing errors, scheduling by time multiplexing, task partitioning, and problems arising from the use of two new language constructs (rendezvous and exception propagation). It is noted that in real-time distributed systems, the accuracy of calculations will be always affected by a time skew and that avionics systems must be designed to tolerate this variation. Care must be also taken to ensure a reasonable bound for the inherent latency of multiprocessor communications. 83/00/00 84A16647

UTTL: ADATLAS - The test language of the future

AUTH: A/ANDERSON, R. E.; B/MCGARVEY, R. L.; C/ZEAFLA, L. A. PAA: C/(AAI Corp., Baltimore, MD) In: AUTOTESTCON '81; Proceedings of the Conference, Orlando, FL, October 19-21, 1981. (A83-10726 01-59) New York, Institute of Electrical and Electronics Engineers, Inc., 1981, p. 94-101.

ABS: ADA is the new DOD standard High Order Language intended for use in embedded computer systems. This paper proposes that it could be more cost effective to write unit under test (UUT) programs in ADA instead of ATLAS, the currently accepted standard test language. This could be accomplished by combining the test oriented features of ATLAS with the common programming features of ADA, specifying a standard ADA package that provides the test oriented functions. 81/00/00 83A10741

UTTL: ANSI Ada and the UK M-CHAPSE

AUTH: A/BARNES, J. G. P. CORP: European Space Agency, Paris (France). In ESA Software Eng. p 243-250 (SEE N84-14729 05-61)

ABS: The major changes to Ada resulting from the ANSI standardization process are summarized. The M-CHAPSE project to provide the foundation for a professional programming support environment for Ada and CHILL is described. The M-CHAPSE objectives are: to provide host-target development systems with source debugging on the host; to provide consistent and convenient user friendly interfaces; to provide an open ended environment which accommodates further tools; to provide a foundation which supports all aspects of the entire system life cycle; to provide a portable system which can be rehosted and retargetted at reasonable cost; and to provide a secure and reasonably efficient system. 83/08/00 84N14760

UTTL: Software engineering

AUTH: A/BATTRICK, B.; B/ROLFE, E. J. CORP: European Space Agency, Paris (France). Proc. of ESA/ESTEC Sem., Noordwijk, Netherlands, 11-14 Oct. 1983

RPT#: ESA-SP-199 ISSN-0379-6566 83/08/00 84N14729

UTTL: Ada (trademark) as a program design language: Have the major issues been addressed and answered?

AUTH: A/BLASEWITZ, R. M. CORP: RCA Government Systems Div., Moorestown, N. J. CSS: (Missile and Surface Radar.) In Army Communications-Electronics Command Proc. of the 2nd Ann. Conf. on Ada (Trademark) Technol. p 111-114 (SEE N84-30745 20-61)

ABS: Department of Defense requirements to use the higher order language Ada will create challenges to developers of military software that encompass these major concerns: (1) developing a core of Ada software personnel, (2) achieving productivity and software gains that have been targeted as Ada life-cycle objectives, and (3) transitioning to a language that embodies a capability to express software solutions eloquently, clearly, reliably and efficiently. Ada is more than a programming language, it is the basis for a modern perspective of software design and engineering. The IEEE working group on Ada as a PDL has been addressing the issues involved with the use of Ada as a design mechanism for nearly two years. This working group has recently generated a draft guideline that addresses the key issues. The extent of

industry's involvement with Ada PDLs and the status and final form of the IEEE product will substantially impact both the acceptance of the Ada language and the efficiency and correctness of its use.
RPT#: AD-P003430 84/03/00 84N30762

UTTL: Ada (Trademark) as a program design language. A rational approach to transitioning industry to the world of Ada through a program design language criteria
AUTH: A/BLASEWITZ, R. M. CORP: RCA Government Systems Div., Moorestown, N. J. CSS: (Missile and Surface Radar.) In ASD Proc. Papers of the 2nd AFSC Avionics Standardization Conf., Vol. 1 p 509-513 (SEE N84-31121 21-06)
ABS: The Department of Defense requirements to use the higher order language Ada by the mid-1980s will create challenges to developers of military software that encompass two major concerns: developing a core of Ada software personnel, and achieving productivity and software quality gains that have been targeted as Ada life cycle objectives. Because of recent government direction to use Ada-based PDLs, many organizations are developing prototype Ada-based design methods. The IEEE working group on Ada as a PDL is working on guidelines for the use of Ada-based design languages. The guidelines will include recommendations reflecting the current state of the art as well as alternative approaches in order to preserve good practice. The extent of industry's involvement with Ada PDLs, and also the status of the IEEE guidelines, may substantially impact both the acceptance of the Ada language and the efficiency of its use.
RPT#: AD-P003556 82/11/00 84N31160

UTTL: Seeding the Ada (trademark) software components industry
AUTH: A/BOWLES, K. CORP: Telesoft, San Diego, Calif. In Army Communications-Electronics Command Proc. of the 2nd Ann. Conf. on Ada (Trademark) Technol. p 125-128 (SEE N84-30745 20-61)
ABS: The principal aim of the Ada effort is economic - particularly the enhancement of designer/programmer productivity in all parts of the software life-cycle. A shift in system design practice to widespread use of off-the-shelf large scale Ada software components would result in productivity gains exceeding a factor of ten - far more than likely to result from use of productivity enhancing software tools. To achieve widespread use of off-the-shelf Ada components requires establishment of a software components industry, and a shift in attitudes about education of system designers to use Ada. This paper reviews progress to date.
RPT#: AD-P003432 84/03/00 84N30764

UTTL: Ada - A good start, an exciting future
AUTH: A/BRAUN, C. L. PAA: A/(SofTech, Inc., Waltham, MA) Defense Electronics (ISSN 0278-3479), vol. 17, July 1985, p. 105, 106.
ABS: An evaluation is made of the utility of the U.S. Department of Defense standard computer language, Ada, at the current stage of its development, and the further performance improvements that may be obtained in the course of its development history. It is noted that the full advantages that accrue to the comprehensive use of a single high order language by most Pentagon contractors will only begin to be realized as entirely new software-intensive projects are conceived; several major systems now entering service antedate Ada. It is not expected that fourth-generation high order languages incorporating refinements beyond those embodied in Ada will be ready in less than 10 years, which was the development period length for Ada itself. 85/07/00 85A41549

UTTL: Ada (trademark) Programming Support Environment (APSE) Evaluation and Validation (E and V) workshop report held at Airlie, Virginia on 2-6 April 1984
AUTH: A/CASTOR, V. L.; B/ANDERSON, C. M.; C/LINDQUIST, T. E.; D/KRAMER, J. F., JR. CORP: Institute for Defense Analyses, Alexandria, Va. Workshop held at Airlie, Va., 2-6 Apr. 1984
ABS: The Evaluation & Validation (E&V) Task has been established by the AJPO to develop technology necessary to assess the quality of software tools that will be included in an Ada Programming Support Environment (APSE). An E&V Workshop convened April 2 to 6, 1984, consisted of government and industry representatives who developed draft documents in three E&V subject areas--Task Recommendations, Task Requirements, and the APSE Evaluation Reference Manual. This report contains an account of the plenary sessions and the papers that address the three subject areas under discussion during this workshop.
RPT#: AD-A156667 AD-E500721 IDA-M-34 IDA/HQ-84-29295 84/12/00 86N10829

UTTL: Evaluation and Validation (E/V) team public report, volume 1
AUTH: A/CASTOR, V. L. CORP: Air Force Wright Aeronautical Labs., Wright-Patterson AFB, Ohio.
ABS: The initial activities and accomplishments of the Evaluation and Validation (E&V) Team are reported. The purpose of the E&V Task, which is sponsored by the Ada Joint Program Office (AJPO), is to develop the techniques and tools which will provide a capability to perform assessment of Ada Programming Support Environments (APSEs) and to determine conformance of APSEs to the Common APSE Interface Set (CAIS). As this technology is developed, it is being made available to DOD components, industry and academia. As with all Ada-related activities, the widest possible participation in the E&V Task is encouraged.
RPT#: AD-A153609 AFWAL-TR-85-1016-VOL-1 84/11/30 85N29584

UTTL: Teaching Ada (trademark) at the US Military Academy
AUTH: A/COGAN, K. J. CORP: Military Academy, West Point, N. Y. CSS: (Dept. of Geography and Computer Science.) In Army Communications-Electronics Command Proc. of the 2nd Ann. Conf. on Ada (Trademark) Technol. p 31-34 (SEE N84-30745 20-61)
ABS: A five year history of teaching Ada* with the NYU Ada/Ed translator has evolved into an effective methodology for teaching top-down engineering design simultaneously with a bottom-up presentation of the Ada grammar. With emphasis on embedded hardware systems, students are confronted with successively more difficult design problems which must be written and executed on a VAX-11/780. Exposed to the Ada features of packages, concurrency, generics, and exception handling, students design, write and execute an extensive term project simulating a real-time embedded system using Ada. Projects approach the 1000 lines of source code limitation of the translator. Reusability of code is stressed by importing a previous year's package when feasible.
RPT#: AD-P003418 84/03/00 84N30750

UTTL: Experimenting with the Rolm Ada language workshop for specifying and designing
AUTH: A/COUSERGUE, M.; B/PELLIZZARI, M. CORP: Centre National d'Etudes Spatiales, Toulouse (France). Presented at DESS Training Session, Toulouse
ABS: The utilization of Ada Language in specifying and designing large real time computer programs for space studies applications is discussed. Comparison with languages such as SADT for specifications and PDL for designing, are made. The Ada development environment is presented theoretically and as a quality analysis. It is shown that Ada is not acceptable for program specifications, but quite adequate for designing.
84/06/00 85N12607

UTTL: The Ada (Trademark) run-time environment
AUTH: A/CROSS, J. K. CORP: Sperry Univac, St. Paul, Minn. CSS: (Defense Systems Div.) In ASD Proc. Papers of the 2nd AFSC Avionics Standardization Conf., Vol. 1 p 533-538 (SEE N84-31121 21-06)
ABS: The requirements on an Ada run-time environment are surprisingly few and straightforward. The free choices left up to the implementors of a run-time environment are many and significant. These requirements and freedoms are enumerated and discussed, and the importance of these issues to the success of an Ada software system is described.
RPT#: AD-P003558 82/11/00 84N31162

UTTL: In praise of procedures
AUTH: A/CURRIE, I. F. CORP: Royal Signals and Radar Establishment, Malvern (England).
ABS: The use of procedures vary greatly from one programming language to another. These variations are discussed and the use of procedures in a very general fashion is argued; in particular, procedures are an obvious vehicle to provide the data abstraction and encapsulation given in a very limited form by other language constructs such as ADA packages. The implementation of these general procedural values is also discussed with reference to the Flex computer.
RPT#: RSRE-3499 BR85316 82/07/00 83N18300

UTTL: Using Ada for a distributed, fault tolerant system
AUTH: A/DEWOLF, J. B.; B/SODANO, N. M.; C/WHITTREDGE, R. S. PAA: C/(Charles Stark Draper Laboratory, Inc., Cambridge, MA) CORP: Draper (Charles Stark) Lab., Inc., Cambridge, Mass. IN: Digital Avionics Systems Conference, 6th, Baltimore, MD, December 3-6, 1984, Proceedings (A85-17801 06-01). New York, American Institute of Aeronautics and Astronautics, 1984, p. 477-484.
ABS: It is pointed out that advanced avionics applications increasingly require underlying machine architectures which are damage and fault tolerant, and which provide access to distributed sensors, effectors and high-throughput computational resources. The Advanced Information Processing System (AIPS), sponsored by NASA, is to provide an architecture which can meet the

considered requirements. Ada was selected for implementing the AIPS system software. Advantages of Ada are related to its provisions for real-time programming, error detection, modularity and separate compilation, and standardization and portability. Chief drawbacks of this language are currently limited availability and maturity of language implementations, and limited experience in applying the language to real-time applications. The present investigation is concerned with current plans for employing Ada in the design of the software for AIPS. Attention is given to an overview of AIPS, AIPS software services, and representative design issues in each of four major software categories.

RPT#: AIAA PAPER 84-2703 84/00/00 85A17873

UTTL: A host-target programming support environment for the production of high-quality real-time systems

AUTH: A/DOWLING, E. J.; B/AVIS, B. E. CORP: Ferranti Computer Systems Ltd., Cwmbran (England). In ESA Software Eng. p 191-198 (SEE N84-14729 05-61)

ABS: Hardware and software components of an environment used for the production of large, real time systems are discussed, together with the reasons behind the choices made. Many software tools come directly from the host system (a DEC VAX with VMS) but tools for software (configuration) management and software verification were developed. Ada and Ada program support environments are contrasted with features of the environment currently used. 83/08/00 84N14753

UTTL: Diana reference manual, revision 3

AUTH: A/EVANS, A., JR.; B/BUTLER, K. J. CORP: Tartan Labs., Inc., Pittsburgh, Pa.

ABS: This document describes Diana, a Descriptive Intermediate Attributed Notation for Ada, being both an introduction and reference manual for it. Diana is an abstract data type such that each object of the type is a representation of an intermediate form of an Ada program. Although the initial uses of this form were for communication between the Front and Back Ends of an Ada compiler, it is also intended to be suitable for use with other tools in an Ada programming environment. Diana resulted from a merger of the best properties of two earlier similar intermediate forms: TCOL and AIDA.

RPT#: AD-A128232 TL-83-4 83/02/28 83N35684

UTTL: Software development methodologies and Ada. Ada methodologies: Concepts and requirements, Ada methodology questionnaire summary, comparing software design methods for Ada: A study plan

AUTH: A/FREEMAN, P.; B/WASSERMAN, A. I. CORP: California Univ., Irvine.

ABS: This document rationalizes the need for the use of coherent software development methodologies in conjunction with Ada and its programming support environments (APSE's) and describes the characteristics that such methodologies should possess. It is recognized that software development, particularly for embedded systems, is increasingly done in the context of overall systems development, including hardware and environmental factors. While there is a strong need for integrated systems engineering, this document focuses on the software issues only. Emphasis is thus given to the process by which software is developed for Ada applications, not just with the language or its automated support environment. The development activity yields a collection of work products (including source and object versions of Ada programs). These work products are valuable not only through the development phase, but also through the entire lifetime of the system as modifications and enhancements are made to the system.

RPT#: AD-A123710 82/11/00 83N26541

UTTL: Parameterized programming

AUTH: A/GOGUEN, J. A. PAA: A/(SRI International, Menlo Park; Stanford University, Stanford, CA) IEEE Transactions on Software Engineering (ISSN 0098-5589), vol. SE-10, Sept. 1984, p. 528-543.

ABS: The present investigation is concerned with a technique called 'parameterized programming' which can greatly extend the opportunities for reusing software modules. The basic idea of parameterized programming is to maximize program reuse by storing programs in as general a form as possible. The implementation of this concept requires a suitable notion of a parameterized module, along with the capability for instantiating the parameters of such modules, and the capability for encapsulating existing code into modules. Language features to support parameterized programming are considered along with an illustration of parameterized programming with some simple examples written in the OBJ programming system now under development. Attention is given to the hierarchical structure, rewrite rules, details regarding parameterized modules, and the denotational semantics of OBJ. 84/09/00 85A11097

UTTL: Ada compiler validation summary report: ROLM Ada compiler, version 4.42 V-002
AUTH: A/GOODENOUGH, J. B. CORP: Softech, Inc., Waltham, Mass.
ABS: This report describes the results of the validation effort for the following Ada translator: Host Machine: ROLM MSE/800, Data General MV/4000, MV/6000, MV/8000, and MV/10000; Operating System: AOS/VS-Ada 2.03; Host Disk System: 2.96 megabyte drives; Target Machine: ROLM MSE/800, Data General MV/4000, MV/6000, MV/8000, and MV/10000; Operating System: AOS/VS-Ada 2.03; Language Version; ANSI/MIL-STD-1815A Ada; Translator Version: 4.42; and Validator Version: 1.1 (March 4, 1983). Testing of this translator was conducted by SofTech, Inc. The purpose of this report is to document the results of the testing performed on the translator, and in particular, to: identify any language constructs, supported by the translator that do not conform to the Ada standard; identify any unsupported language constructs required by the Ada standard; and describe implementation-dependent behavior allowed by the standard.
RPT#: AD-A136732 83/05/12 84N20199

UTTL: Ada (trademark) design language concerns
AUTH: A/GRAU, J. K.; B/COMER, E. R. CORP: Harris Corp., Melbourne, Fla. In Army Communications-Electronics Command Proc. of the 2nd Ann. Conf. on Ada (Trademark) Technol. p 115-124 (SEE N84-30745 20-61)
ABS: This paper examines key language concerns regarding Ada Design Languages (DL's) in regard to: life cycle applicability; the information expressed by an Ada DL; relationship of an Ada DL to the Ada language; extensions of the Ada language through structured commentary and annotation; and the relationship between methodology and Ada Design Language. An assessment is made of the relative maturity of Ada DL's and of the obstacles to successful development of an Ada DL standard.
RPT#: AD-P003431 84/03/00 84N30763

UTTL: An advanced host-target environment for the military computer family
AUTH: A/HART, H.; B/HART, R.; C/MUENNICHOW, I. CORP: TRW, Inc., Redondo Beach, Calif. In Army Communications-Electronics Command Proc. of the 2nd Ann. Conf. on Ada (Trademark) Technol. p 89-101 (SEE N84-30745 20-61)
ABS: As part of the Military Computer Family Operating System (MCFOS) project, extensions to the Ada Language System (ALS) are being constructed which allow software for the MCF computers to be developed and tested in a host/target environment. These extensions are collectively known as the ALSE. ALS facilities are used for editing, compiling, linking, and exploring Ada programs, while ALSE facilities are used to download the software into a connected MCF computer and execute the software on the MCF, thus providing state-of-the-art high level debugging and performance monitoring facilities in an embedded target environment. This paper describes the components of the ALSE from a user viewpoint, concentrating on how an applications programmer would use MCFOS and the Extended Ada Language System to develop software.
RPT#: AD-P003428 84/03/00 84N30760

UTTL: An automated methodology development
AUTH: A/HAWLEY, L. R. PAA: A/(California Institute of Technology, Jet Propulsion Laboratory, Pasadena, CA) CORP: Jet Propulsion Lab., California Inst. of Tech., Pasadena. IN: Simulation in Ada; Proceedings of the Eastern Simulation Conference, Norfolk, VA, March 3-8, 1985 (A85-34127 15-61). San Diego, CA, Society for Computer Simulation, 1985, p. 1-5. Army-sponsored research.
ABS: The design methodology employed in testing the applicability of Ada in large-scale combat simulations is described. Ada was considered as a substitute for FORTRAN to lower life cycle costs and ease the program development efforts. An object-oriented approach was taken, which featured definitions of military targets, the capability of manipulating their condition in real-time, and one-to-one correlation between the object states and real world states. The simulation design process was automated by the problem statement language (PSL)/problem statement analyzer (PSA). The PSL/PSA system accessed the problem data base directly to enhance the code efficiency by, e.g., eliminating non-used subroutines, and provided for automated report generation, besides allowing for functional and interface descriptions. The ways in which the methodology satisfied the responsiveness, reliability, transportability, modifiability, timeliness and efficiency goals are discussed. 85/00/00 85A34128

UTTL: An expansive view of reusable software
AUTH: A/HOROWITZ, E.; B/MUNSON, J. B. PAA: A/(Southern California, University, Los Angeles, CA); B/(System Development Corp., Camarillo, CA) IEEE Transactions on Software Engineering (ISSN 0098-5589), vol. SE-10, Sept. 1984, p. 477-487.
ABS: Developments related to software development have not kept pace with advances related to computer hardware, and the cost of software development has become

expensive. One of the reasons for this phenomenon involves a lack of significant improvements made in the productivity of software development work. The present investigation is concerned with a concept which has the potential for increasing software productivity. That concept is now known by the expression 'reusable software'. The advantages of an employment of reusable software are related to the observation that much of the code of one system is virtually identical to code which was previously written. In one example, it was found that 40-60 percent of actual program code was repeated in more than one application. However, previous attempts to utilize reusability concepts have not been entirely successful. The current investigation has the objective to examine reusability in its many forms and to discuss their merits, primarily in the domain of programming customized software systems. 84/09/00 85A11095

UTTL: Preliminary program manager's guide to Ada
AUTH: A/HOWE, R. G.; B/BYRNE, W. E.; C/GRUND, E. C.; D/HILLIARD, R. F. I. ; E/MUNCK, R. G. CORP: Mitre Corp., Bedford, Mass.
ABS: This draft guide provides current information that should help a Program Manager in making decisions relative to the use of Ada. It discusses pertinent Air Force and DoD policy, effects of Ada on contractual documentation, and steps that must be taken to apply Ada. This guide identifies benefits and inherent risks of using Ada, and Program Office initiatives needed to control risk. It cites factors that will affect programmer training, software cost and schedule estimation, design, and configuration management. As background information, the basic features of Ada and the software needed to support programming in Ada are described. The guide will be issued after review.
RPT#: AD-A140347 WP-25012 ESD-TR-83-255 84/02/00 84N26457

UTTL: Concept paper for the development of a DOD Ada (trademark) software engineering education and training plan
AUTH: A/JORDAN, P. R.; B/MCDONALD, C. W.; C/SCHAAR, B. CORP: Institute for Defense Analyses, Alexandria, Va.
ABS: The Ada Joint Program Office (AJPO) was established in December 1980, to manage the Department of Defense (DOD) efforts to implement, introduce, and provide life-cycle support for Ada. As part of this charter, it is the role of the AJPO to address Ada education and training. The goal of this document is to set forth the concepts necessary for Ada software engineering education and training. These concepts will result in an effective use of Ada in the shortest time possible to realize cost savings and achieve reliability and adaptability in computer software development. The full potential of Ada cannot be realized without appropriate education and training.
RPT#: AD-A148774 AD-E500686 IDA-M-7 IDA/HQ-84-28940 84/11/00 85N17592

UTTL: An evaluation of the needs and requirements for the establishment of an Ada (trade name) liaison organization
AUTH: A/JORSTAD, N. D.; B/LEE, J. A. N.; C/LOMONACO, S. J., JR. CORP: Institute for Defense Analyses, Alexandria, Va. CSS: (Science and Technology Div.)
ABS: This report examines the needs of the Ada Joint Program Office (AJPO) in the establishment of an organization (to be called the Ada Liaison Organization (ALO)) for the "sustenance' of the Ada language following the acceptance of the language as a National standard and during the early stages of implementation, expansion of applicability, and development of supporting systems. It is proposed, in view of the number of overlapping needs of various responsibilities, that such an organization be formed as a single unit so as to maintain coordination, but with distinct subentities interacting with the differing internal and external agencies which have cognizance of some phase of the Ada activity. The ultimate responsibility for the continued support and development of Ada remains with AJPO. The Ada Liaison Organization will provide recommendations for the administration of the Military Standard, the American National Standard and other related standards or specifications. The proposed charter of the ALO is presented in the Appendix.
RPT#: AD-A122286 AD-E500546 IDA-P-1681 IDA/HQ-82-24869 82/09/00 83N21825

UTTL: APSE database user scenario
AUTH: A/KEAN, E. S. CORP: Rome Air Development Center, Griffiss AFB, N.Y. In ASD Proc. Papers of the 2nd AFSC Avionics Std. Conf., Vol. 2 p 1085-1096 (SEE N84-31165 21-06) Proc. held in Dayton, Ohio, 30 Nov. - 2 Dec. 1982
ABS: The Database facilities provided in the Ada (1) Integrated Environment are an integral part of the KAPSE/MAPSE design described in the STONEMAN document. Project management and configuration control facilities are essential elements in the design. They are provided through primitives in the database management system incorporated in the AIE

KAPSE/Database. Large and small scale system development can be easily managed by manipulating the primitives to meet user requirements. The users may range from project managers, software designers, programmers, and quality assurance personnel to administrative support personnel. These users have very different functional requirements and make various demands on the database that will require a versatile and efficient database system. This paper describes the users interfaces and capabilities provided to AIE users during the development cycle of a software system.
RPT#: AD-P003593 82/11/00 84N31198

UTTL: Will Ada succeed?
AUTH: A/KLEIN, D. PAA: A/(Rolm Corp., San Jose, CA) Defense Electronics (ISSN 0278-3479), vol. 16, Dec. 1984, p. 105, 106, 108.
ABS: The U.S. Department of Defense standard programming language, Ada, offers enhanced software reliability, transportability, simplified maintenance, and life cycle cost reductions. Its critics have cited problems, however, in compiler technology immaturity and excessively severe training requirements. An evaluation is presently made of the status of Ada software implementation by the U.S. Armed Services, as well as the comparative advantages of Ada, in light of past experience with such languages as Fortran and Cobol. 84/12/00 85A34200

UTTL: A proposal for standard basic functions in Ada
AUTH: A/KOK, J.; B/SYMM, G. T. PAA: A/(Centrum voor Wiskunde en Informatica, Amsterdam) CORP: National Physical Lab., Teddington (England). CSS: (Div. of Information Technology and Computing.)
ABS: A standard basic mathematical functions package for scientific computation in Ada is proposed. The package is transportable to machines with different floating-point types and its availability enhances the portability of numerical software.
RPT#: NPL-DITC-45/84 ISSN-0262-5369 CWI-NM-R8407 PB85-118701 84/06/00 85N14584

UTTL: Ada status and outlook
AUTH: A/KRAMER, J. F., JR. CORP: Ada Joint Program Office, Arlington, Va. In AGARD Software for Avionics 6 p (SEE N83-22112 12-01)
ABS: The Ada programming language is discussed. 83/01/00 83N22123

UTTL: An Ada (trademark) network: A real-time distributed computer system
AUTH: A/LANE, D. S.; B/HULING, G.; C/BARDIN, B. M. CORP: Hughes Aircraft Co., Fullerton, Calif. In Army Communications-Electronics Command Proc. of the 2nd Ann. Conf. on Ada (Trademark) Technol. p 55-61 (SEE N84-30745 20-61)
ABS: This paper reports on a prototype real-time distributed computer system which will be implemented in Ada. The goals of the Ada network are to support transparent distribution of application software and incremental growth, and to provide fault tolerant capabilities. Some of the motivations and methods for distributing software in a local network of communicating processors are discussed, in addition to the hardware configuration and software development facilities. The model of distributed Ada programs is then described. After the prototype software is implemented, the project will focus on assessing the performance characteristics of the network: specifically, on distribution, executive software, and Ada language overheads.
RPT#: AD-P003423 84/03/00 84N30755

UTTL: An optimizing table driven code generator for an Ada compiler
AUTH: A/LEONARD, T. M. PAA: A/(Florida State University, Tallahassee, FL) IN: SOUTHEASTCON '83; Proceedings of the Region 3 Conference and Exhibit, Orlando, FL, April 11-13, 1983 (A85-28101 11-33). New York, Institute of Electrical and Electronics Engineers, Inc., 1983, p. 81-85.
ABS: The code generator presented in this paper is part of an Ada cross-compiler, hosted on a CDC Cyber 6000 series computer and targeted for the Z8002 microprocessor, presently being developed at Florida State University. It is a table driven code generator automatically constructed from an attribute grammar description of a translation from an intermediate language of the compiler to a Z8002 assembly language. The code generator tables are automatically derived from the grammar using an attribute grammar processor and an LALR(1) parser generator, thus automating much of the code generation process. 83/00/00 85A28103

UTTL: A specification technique for the common APSE (Ada programming support environments) interface set
AUTH: A/LINDQUIST, T. E.; B/FACEMIRE, J. L.; C/KAFURA, D. G. CORP: Virginia Polytechnic Inst. and State Univ., Blacksburg. CSS: (Dept. of Computer Science.)
ABS: This report demonstrates an approach to specifying kernel Ada support environment interface components.

The objectives are to provide a mechanism which allows building a complete enough specification for validation, an understandable specification, and one that is relatively easy to construct. In meeting these objectives, an Abstract Machine approach has been modified and applied to functional description of kernel operations. After motivating an explaining the approach, the paper exemplifies its utility. Interactions among kernel operations and pragmatic implementation limits, which are other needed parts of a specification, are also discussed.
RPT#: AD-A140889 CS84004-R 84/04/00 84N27479

UTTL: Design of Ada systems yielding reusable components - An approach using structured algebraic specification
AUTH: A/LITVINTCHOUK, S. D.; B/MATSUMOTO, A. S. PAA: A/(Mitre Corp., Bedford, MA); B/(ITT Advanced Technology Center, Stratford, CT) IEEE Transactions on Software Engineering (ISSN 0098-5589), vol. SE-10, Sept. 1984, p. 544-551.
ABS: Experience with design of Ada software has indicated that a methodology, based on formal algebra, can be developed which integrates the design and management of reusable components with Ada systems design. The methodology requires the use of a specification language, also based on formal algebra, to extend Ada's expressive power for this purpose. It is shown that certain requirements for the use of Ada packages which cannot be expressed in Ada can be expressed in algebraic specification languages, and that such specifications can then be implemented in Ada.
84/09/00 85A11098

UTTL: Annotation language design for Ada (ANNA)
AUTH: A/LUCKHAM, D. C. CORP: Stanford Univ., Calif. CSS: (Computer Systems Lab.)
ABS: This interim report covers research work on Annotation language design for Ada. The major goal of this research was the design and development of programming tools that may be incorporated into an Ada Programming Support Environment during the mid-1980's time frame. Since Ada is a very advanced language containing many essential new features such as tasking, and standard Ada tools such as compilers do not yet exist, the research has been structured so as to approach the major goal by first studying the error detection problem for subsets of Ada corresponding to already highly used languages such as Pascal. The error detection problem as an important starting point because this attempts to analyse programs for common errors without assuming that the programs have accompanying annotations. At the start of this project no formal annotation language for Ada existed. The second phase of the research effort was to design an annotation language for Ada, called ANNA. This would provide a basis for verification of Ada programs in general. This second report deals with the work on the design of ANNA.
RPT#: AD-A140452 RADC-TR-83-298 84/01/00 84N26336

UTTL: Ada advanced error detector
AUTH: A/LUCKHAM, D. C. CORP: Stanford Univ., Calif. CSS: (Computer Systems Lab.)
ABS: This is the final technical report on a project entitled "Ada Advanced Error Detector.' The purpose of this project was to study techniques of detecting common runtime errors in sequential Ada at compile-time using verification techniques, high level annotation languages, and runtime detection of deadness errors in Ada tasking. This work has resulted in a working prototype implementation of a system for detecting and diagnosing tasking errors.
RPT#: AD-A140273 RADC-TR-83-299 84/01/00 84N26334

UTTL: Software library: A reusable software issue
AUTH: A/METCALF, S. G. CORP: Naval Postgraduate School, Monterey, Calif.
ABS: This thesis presents a conceptual view of a reusable Software Library. Issues concerning the software crisis and its subsequent impact on software development are reviewed. The traditional library is described for the purpose of comparison with the Software Library. A particular example of the Software Library, the Program Library, is described as a prototype of a reusable library. A hierarchical structure for a Program Library is discussed as an approach to making the library entities easily accessible and retrievable. The role of application generators in the Program Library is described. The special features of Ada that support programming libraries are described. Finally, noncode products in the Software Library are discussed.
RPT#: AD-A150722 84/06/00 85N24816

UTTL: Flowcharting with D-charts
AUTH: A/MEYER, D. CORP: Jet Propulsion Lab., California Inst. of Tech., Pasadena.
ABS: A D-Chart is a style of flowchart using control symbols highly appropriate to modern structured programming languages. The intent of a D-Chart is to provide a clear and concise one-for-one mapping of control symbols to high-level language constructs for

purposes of design and documentation. The notation lends itself to both high-level and code-level algorithmic description. The various issues that may arise when representing, in D-Chart style, algorithms expressed in the more popular high-level languages are addressed. In particular, the peculiarities of mapping control constructs for Ada, PASCAL, FORTRAN 77, C, PL/I, Jovial J73, HAL/S, and Algol are discussed.
RPT#: NASA-CR-175641 JPL-PUB-84-99 NAS 1.26:175641 85/01/15 85N24806

UTTL: Impact of ADA on the software life cycle
AUTH: A/MICKEL, S. B. PAA: A/(General Electric Co., Sunnyvale, CA) IN: Computers in Aerospace Conference, 4th, Hartford, CT, October 24-26, 1983, Collection of Technical Papers (A84-10001 01-59). New York, American Institute of Aeronautics and Astronautics, 1983, p. 200-204.
ABS: The introduction of the ADA programming language to the software development process is discussed. The software life cycle is outlined and illustrated with block diagrams, and it is shown that ADA, because of its flexibility, will affect almost every aspect of the cycle, especially the phase transitions (where different languages meet using present techniques), the design methodology, software reusability, and programmer training. While experience with ADA is required to maximize the benefits it offers, some preliminary recommendations (for the next few years) can be made, including revision of the military standard specifications, types, and forms; modification of software-acquisition practices to promote reusability; standard nomenclature for embedded systems; and the development of training programs based on sets of graduated examples.
RPT#: AIAA PAPER 83-2362 83/00/00 84A10030

UTTL: Adaptive robust and nonparmetric procedures with application to communications, radar, sonar and array signal processing
AUTH: A/MIHAJLOVIC, R. A. CORP: Polytechnic Inst. of New York, Brooklyn.
ABS: The qualitative aspects of robustness in the signal detection framework is addressed. Local and global robustness properties of the detector are investigated and various approaches to the quantitative and qualitative characterization of the robustness are introduced. The measures of robustness are easy to calculate and provide valuable information about performance of the detector in a severe noise environment. Of particular interest to the designer is the efficacy robustness curve (ERC) and the efficacy loss density curve (ELDC) which provide information pertaining to two conflicting performance criteria: insensitivity to impulsive noise contamination and efficacy (power). Based on the proposed performance measures and descriptors some detectors, such as, the linear or the quadratic detector are shown to be nonrobust. Their comparison in adverse noise conditions with what is considered as robust detectors, in the framework of the presented theory, is improper. 82/00/00 83N13327

UTTL: Preliminary design and implementation of a method for validating evolving Ada compilers
AUTH: A/MILLER, E. D., JR. CORP: Air Force Inst. of Tech., Wright-Patterson AFB, Ohio. CSS: (School of Engineering.)
ABS: This project consisted of a preliminary design and a partial implementation of a tool which modifies the existing Ada Compiler Validation Capability (ACVC) test set so it can be used to test evolving Ada compilers. The project evaluated the feasibility of repackaging each of test classes found in the ACVC and suggested methods for repackaging the tests. The tool developed uses a table-driven parser which parses the July 1980 proposed standard. In uses output from the parser to generate a representation of a test program. Once the representation is developed, unsupported language-features are removed from it. The remaining representation is output as a valid test program.
RPT#: AD-A127333 AFIT/GCS/MA/83M-1 83/03/00 83N31338

UTTL: Ada-Europe guidelines for Ada compiler specifications and selection
AUTH: A/NISSEN, J. C. D.; B/WICHMANN, B. A.; C/DOWLING, T. ; D/GOLDSACK, S.; E/JEANROD, H.; F/MONTGOMERY, A.; G/PIERCE, R.; H/REFICE, M.; I/THETHANG, N.; J/TAFVELIN, S. CORP: National Physical Lab., Teddington (England). CSS: (Div. of Information Technology and Computing.)
ABS: The Ada language reference manual defines the language rather than indicating a list of the desirable properties of an implementation of the language. The characteristics of an implementation that should be taken into account in the specification or selection of an Ada compiler are listed.
RPT#: NPL-DITC-10/82 ISSN-0262-5369 82/10/00 83N18287

UTTL: Ada-Europe guidelines for the portability of Ada programs, second edition
AUTH: A/NISSEN, J. C. D.; B/WALLIS, P.; C/WICHMANN, B. A.; D/BARNES, J.; E/BRIGGS, J.; F/DOWLING, T.; G/GOLDSACK, S.; H/JEANROND, H.; I/MONTANEGRO, C.; J/MONTGOMERY, A. CORP: National Physical Lab., Teddington (England). CSS: (Div. of Information Technology and Computing.)
ABS: A users guide for designing and coding portable Ada programs is supplied. Layout and numbering follow the Ada language reference manual. Tasks, program structure, compilation issues, exceptions, generic program units, representation specifications, implementation dependent features, input/output routines, lexical elements, declarations, types, names, expressions, statements, subprograms, packages, and visibility rules are covered. Guidelines are classified as mandatory, recommended, or suggested.
RPT#: NPL-DITC-27/83 ISSN-0262-5369 83/07/00 84N12739

UTTL: Kernel ADA Programming Support Environment (KAPSE) interface team: Public report, volume 2
AUTH: A/OBERNDORF, P. A. CORP: Naval Ocean Systems Center, San Diego, Calif.
ABS: The continuing activities of the Kernel Ada Programming Support Environments (KAPSE) interface team and its industry/academia auxiliary are reported. (Ada is a recent, DOD-developed programming language.) The Ada Joint Program Office (AJPO)-sponsored effort will ensure the interoperability and transportability of tools and data bases among different KAPSE implementations. The effort is the result of a Memorandum of Agreement (MOA) among the three services directing the establishment of an evaluation team, chaired by the Navy, to identify and establish KAPSE interface standards. As with previous ADA-related developments, the widest possible participation is being encouraged to create a broad base of experience and acceptance in industry, academia, and the DOD.
RPT#: AD-A123136 NOSC/TD-552-VOL-2 82/10/28 83N24198

UTTL: Mapping Ada onto a simple virtual machine
AUTH: A/OCONNELL, S. PAA: A/(Florida State University, Tallahassee, FL) IN: SOUTHEASTCON '83; Proceedings of the Region 3 Conference and Exhibit, Orlando, FL, April 11-13, 1983 (A85-28101 11-33). New York, Institute of Electrical and Electronics Engineers, Inc., 1983, p. 73-77.
ABS: This paper describes the interface between the front and back ends of an Ada compiler for the Z8000 microprocessor. This interface consists of a program called the Traverser, which translates the intermediate language produced by the front end into the second intermediate language (IL2), which is the input to a table-driven code generator. The traverser, the IL2, and the method of code generation have features which make the compiler easily retargetable. The IL2 code is directed toward a virtual machine with a stack based, byte-addressable memory. Target machine-related details in the Traverser are parameterized. The Traverser does all of the storage binding, and interacts with the Code Generator only through a table of IL2 operators. 83/00/00 85A28102

UTTL: Configuration management and the ADA programming support environment
AUTH: A/PULFORD, K. J. CORP: Marconi Avionics Ltd., Boreham Wood (England). In AGARD Software for Avionics 10 p (SEE N83-22112 12-01)
ABS: It is the aim of software development environments to increase the efficiency with which software is produced. One such environment is the ADA programming support environment (APSE) initiated by the U.S. Department of Defence. These environments are a great benefit to programmers making some of their tasks much easier. They also offer great opportunities to monitor and control software development. This in its turn will affect the way that projects are organized and run, and it will affect project personnel jobs to varying extents. The way that projects will be affected by the adoption of an APSE is explored by considering the way that Configuration Management can be implemented in an APSE. 83/01/00 83N22131

UTTL: Artificial intelligence in Ada (trademark): Pattern-directed processing
AUTH: A/REEKER, L. H.; B/KREUTER, J.; C/WAUCHOPE, K. CORP: Tulane Univ., New Orleans, La.
ABS: If the programming language Ada is to be widely used in artificial intelligence applications, it will be necessary to demonstrate to programmers that it can provide superior facilities for use in that domain. One means of doing this is to provide facilities for pattern-directed programming within Ada. This report includes three papers, of which the first is designed to serve as an introduction to pattern-directed programming and to the significance of the two papers that follow. It includes: discussions of artificial intelligence programming and the facilities provided by the Ada language, pattern-directed computation, pattern matching, and parsing. The other two papers deal with the use of Ada for pattern-directed programming. One paper deals with efficient

implementation of pattern matching (within Ada) is important, because pattern matching tends to be inefficient, leading to problems with excessive processing time. Another paper treats extensions of pattern-direction from strings to more general data structures of the sort used in artificial intelligence.
RPT#: AD-A156230 AFHRL-TR-85-12 85/05/00 85N35671

UTTL: Teach Ada (trademark) as the student's first programming language?
AUTH: A/RICHMAN, M. S. CORP: Pennsylvania State Univ., Middletown. In Army Communications-Electronics Command Proc. of the 2nd Ann. Conf. on Ada (Trademark) Technol. p 50-54 (SEE N84-30745 20-61)
ABS: In designing an Ada programming course within our colleges and universities, one of the first issues we must confront is the level of expertise we shall set as prerequisite to the course. Ada is a very rich and complex language. Must the student have experience with some other high order language in order to appreciate Ada? The speaker contends that programming in Ada can be taught in a meaningful way to the neophyte and, in fact, there are decided advantages inherent in learning Ada as a first language. Some suggestions are offered for coping with the size and complexity of Ada.
RPT#: AD-P003422 84/03/00 84N30754

UTTL: Program development with Byron
AUTH: A/SENGUPTA, S.; B/SNYDER, G. PAA: B/(Intermetrics, Inc., Cambridge, MA) IN: NAECON 1984; Proceedings of the National Aerospace and Electronics Conference, Dayton, OH, May 21-25, 1984. Volume 2 (A85-44976 21-01). New York, IEEE, 1984, p. 687-694.
ABS: BYRON, which provides a program development language (PDL) based on the Ada(+) programming language and a set of tools is presented. The tools, which include the BYRON language analyzer, a document generator, and a text formatter, are discussed. The BYRON data flow is shown, as are examples of package specification, algorithm design and implementation, output list of units, and user manual output. 84/00/00 85A45071

UTTL: Ada programming design languages - A report on their status
AUTH: A/SHEFFIELD, J. R.; B/LINDLEY, L. PAA: A/(SofTech, Inc., Dayton, OH); B/(U.S. Navy, Naval Avionics Center, Indianapolis, IN) IN: NAECON 1983; Proceedings of the National Aerospace and Electronics Conference, Dayton, OH, May 17-19, 1983. Volume 2 (A84-16526 05-01). New York, Institute of Electrical and Electronics Engineers, 1983, p. 968-972.
ABS: A survey was performed of Ada-based methodology, tool developments and related project developments which support the concept of using the Ada language as a Programming Design Language (PDL). The survey categorized the work being performed by various companies and individuals, synopsized the work which has been documented to date, and assessed the work relative to its use as a PDL in an Ada framework. The findings of this survey were used to recommend guidelines for an Ada PDL adequate for use on upcoming Navy projects. These guidelines address the popular topic of whether an Ada PPL should be a proper subset of the Ada language, the method for which English narrative may be used to keep the description at the design rather than code level, and the mechanism for adding information to the design not covered by the software engineering concepts embodied in the Ada language proper. 83/00/00 84A16645

UTTL: Applied research in Ada
AUTH: A/STERNE, D. F. PAA: A/(Johns Hopkins University, Laurel, MD) Johns Hopkins APL Technical Digest (ISSN 0270-5214), vol. 5, July-Sept. 1984, p. 266-269.
ABS: Ada is a new state-of-the-art computer programming language developed by the Department of Defense for embedded computer systems. Ada represents a modern approach to reducing software life-cycle costs. In connection with the adoption of Ada by the armed services, certain transition problems arise. For this reason, it was decided to conduct a three-phase program of applied research. The program objectives are discussed, taking into account the progress made in the studies. The objectives of the first phase were to gain hands-on programming experience and to acquire familiarity with the central technical issues surrounding Ada and the Ada Programming Support Environment. The objectives of Phase 2 are to explore the issues of applying Ada to Navy tactical software systems, while Phase 3 is concerned with the development of some specific spinoff products. 84/09/00 85A12219

UTTL: Guidelines for the design of large modular scientific libraries in Ada
AUTH: A/SYMM, G. T.; B/WICHMANN, B. A.; C/KOK, J.; D/WINTER, D. T. PAA: A/(National Physical Lab., Teddington, England); B/(National Physical Lab., Teddington, England) CORP: Center for Mathematics and Computer Science, Amsterdam (Netherlands). CSS: (Dept. of Numerical Mathematics.)

ABS: Solutions for the design and implementation of scientific subroutine libraries in Ada are suggested. Precision, basic mathematical functions, composite data types, information passing, error handling, working space organization, and real time environment are considered.
RPT#: CWI-NM-N8401 B8469329 84/03/00 84N30792

UTTL: Guidelines for the design of large modular scientific libraries in Ada
AUTH: A/SYMM, G. T.; B/WICHMANN, B. A.; C/KOK, J.; D/WINTER, D. T. PAA: C/(Stichting Mathematisch Centrum); D/(Stichting Mathematisch Centrum) CORP: National Physical Lab., Teddington (England). CSS: (Div. of Information Technology and Computing.)
ABS: Guidelines for constructing numerical analysis packages for basic computations, and applications packages, using Ada are presented. Problems related to precision, composite data types, information passing, error handling, working space organization, and real time environment are discussed.
RPT#: NPL-DITC-28/83 ISSN-0262-5369 IR-2 AD-B095226L 83/07/00 84N12740

UTTL: The US Army model Ada (trademark) training curriculum
AUTH: A/TEXEL, P. P. CORP: Softech, Inc., Waltham, Mass. In Army Communications-Electronics Command Proc. of the 2nd Ann. Conf. on Ada (Trademark) Technol. p 5-10 (SEE N84-30745 20-61)
ABS: This paper describes the U.S. Army Model Ada Training Curriculum, developed by Softech, Inc. for the U.S. Army, Ft. Monmouth, N.J. The curriculum consists of individual modules which can be grouped together to form the courses and training plans that best satisfy the needs of specific organizations. The paper describes the modules in terms of content, prerequisites, and status, as of the date of this conference. Finally, the paper addresses how a manager might go about using this curriculum to satisfy the training needs of his organization.
RPT#: AD-P003415 84/03/00 84N30747

UTTL: Configuration management with the Ada (trademark) language
AUTH: A/THALL, R. M. CORP: Softech, Inc., Waltham, Mass. In Army Communications-Electronics Command Proc. of the 2nd Ann. Conf. on Ada (Trademark) Technol. p 11-24 system (SEE N84-30745 20-61)
ABS: Three characteristics of large software projects and five basic configuration management capabilities are identified. The design of the Ada Language System (ALS) is then described in terms of these basic capabilities. The ALS is a computer programming support environment for Ada.
RPT#: AD-P003416 84/03/00 84N30748

UTTL: The Army Ada (trademark) education program
AUTH: A/TURNER, D. J. CORP: Army Communications-Electronics Command, Fort Monmouth, N.J. CSS: (Center for Tactical Computer Systems.) In its Proc. of the 2nd Ann. Conf. on Ada (Trademark) Technol. p 1-4 (SEE N84-30745 20-61)
ABS: In behalf of the U.S. Army, CENTACS is pursuing a comprehensive and aggressive Ada program. An important aspect of that program is the development and transfer of public domain Ada educational and training materials which are focused on the needs of the academic, industrial and government communities. This paper provides an overview of the Army's Ada education and training program and summarizes the products and materials which are being produced under contracts with Softech, Inc., New York University and Jersey City State College.
RPT#: AD-P003414 84/03/00 84N30746

UTTL: An attribute grammar for the semantic analysis of Ada
AUTH: A/UHL, J.; B/DROSSOPOULOU, S.; C/PERSCH, G.; D/GOOS, G.; E/DAUSMANN, M.; F/WINTERSTEIN, G.; G/KIRCHGAESSNER, W. PAA: G/(Karlsruhe, Universitaet, Karlsruhe, West Germany) Research supported by the Bundesamt fuer Wehrtechnik und Beschaffung,. Berlin, Springer-Verlag (Lecture Notes in Computer Science, Volume 139), 1982, 517 p.
ABS: An attribute grammar (AG) that specifies the semantics of the programming language Ada is presented. The development of the AG is traced, together with the applications of semantic analysis within a front-end compiler. Attention is given to attribute value, language elements, entities, syntactic units, and words which change meaning with position in the typed text. The procedure followed in the development of the attribute grammar is explored and the AG is compared with other grammars. Attribute types are defined, as are constants, the STANDARD package, and the start environment. Terminals and nonterminals and their attributes are inventoried, as are the syntactic rules and the semantic functions. 82/00/00 83A45140

UTTL: Ada developments
AUTH: A/VANKATWIJK, J. CORP: Technische Hogeschool, Delft (Netherlands). CSS: (Dept. of Mathematics and Information.)
ABS: Worldwide developments around the programming language Ada are discussed. Structuring and data structures in the Ada subset are described.
RPT#: REPT-84-28 84/00/00 85N29604

UTTL: A concurrent PASCAL implementation on a Rolm 1664
AUTH: A/VANMEURS, W. J. CORP: National Aerospace Lab., Amsterdam (Netherlands). CSS: (Informatics Div.) Presented at 2nd European Rolm Users Group Conf., Wiesbaden, West Germany, 22-23 Mar. 1983
ABS: Concurrent PASCAL (CP) was studied as an intermediate step in the changeover to the Ada language. The transportation of a CP computer and run time environment from a PDP 11/45 to a Rolm 1664 to be used in airborne applications is described. Ways in which CP contributes to reusable software because of its language structure and machine independence are discussed. Transportation of the complete software system (operating system plus application programs) proves to be feasible. As the system being transported is already run, testing of the new implementation can be confined to the virtual machine. The results are increased programmer productivity and program reliability.
RPT#: NLR-MP-83035-U 83/06/12 84N28531

UTTL: The use of Ada in digital flight control systems
AUTH: A/WESTERMEIER, T. F.; B/HANSEN, H. E. PAA: B/(McDonnell Aircraft Co., St. Louis, MO) IN: Guidance, Navigation and Control Conference, Snowmass, CO, August 19-21, 1985, Technical Papers (A85-45876 22-08). New York, AIAA, 1985, p. 597-603.
ABS: A microprocessor-based, parallel-processing flight control system has been built around the F-15 Eagle Dual Control Augmentation System and has been successfully flight tested. The microprocessors are programmed using Ada, the Department of Defense standard high order language. It is widely agreed that Ada has the potential for reducing software life cycle costs through increased programmer productivity. To use Ada and realize the productivity gains, however, the compiler must be reasonably efficient. The use of Ada is discussed, therefore, from these two interrelated standpoints: software productivity and compiler efficiency. The productivity gains and the level of efficiency actually achieved are highlighted.
RPT#: AIAA PAPER 85-1953 85/00/00 85A45939

UTTL: Tutorial material on the real data types in Ada
AUTH: A/WICHMANN, B. A. PAA: A/(European Research Office, London) CORP: National Physical Lab., Teddington (England). CSS: (Div. of Information Technology and Computing.)
ABS: Ada programming language numeric features are outlined, with exercises, for programs familiar with numerical computation but not with Ada. Fixed and floating point; notation for literals; a model of approximate computation; universal expressions; tools for the investigation of numerical accuracy; complex data types; and portability issues are covered.
RPT#: NFL-DITC-34/83 ISSN-0262-5369 84/01/00 84N27458

UTTL: Utilization of Ada as a program design language
AUTH: A/WYLIE, G. J.; B/WATT, T. R. CORP: Naval Postgraduate School, Monterey, Calif.
ABS: In terms of manpower, time and money, the single largest investment that must be made in the acquisition and maintenance of a large and complex computer system is the investment made in software. In response to this situation, DOD began an intensive and comprehensive research and development effort in an attempt to reduce, if not eliminate the inherent problems associated with software system design. The end result of this effort was the creation of the Ada programming language. This thesis will examine the development of the language, focusing attention on the concepts and features which make Ada a potential software crisis solution. These concepts and features will be further examined as to the extent to which they support the utilization of Ada as a program design language.
RPT#: AD-A132244 83/06/00 84N13830

AGARD Lecture Series No.146
Advisory Group for Aerospace Research and Development, NATO
APPLICATION OF ADA* HIGHER ORDER LANGUAGE TO GUIDANCE AND CONTROL
Published May 1986
114 pages

The need to reduce escalating software life-cycle costs is the raison d'être for Ada. Early experience with the language suggests that the promise of increased software productivity can be fulfulled. However, many problems remain: the need for validated and efficient compilers

* Ada is a registered trademark of the US Government (Ada Joint Program Office)

P.T.O.

AGARD-LS-146

ADA (programming languages)
Computer systems programs
Digital computers
Compilers
Operating systems (computers)

AGARD Lecture Series No.146
Advisory Group for Aerospace Research and Development, NATO
APPLICATION OF ADA* HIGHER ORDER LANGUAGE TO GUIDANCE AND CONTROL
Published May 1986
114 pages

The need to reduce escalating software life-cycle costs is the raison d'être for Ada. Early experience with the language suggests that the promise of increased software productivity can be fulfulled. However, many problems remain: the need for validated and efficient compilers

* Ada is a registered trademark of the US Government (Ada Joint Program Office)

P.T.O.

AGARD-LS-146

ADA (programming languages)
Computer systems programm
Digital computers
Compilers
Operating systems (computers)

AGARD Lecture Series No.146
Advisory Group for Aerospace Research and Development, NATO
APPLICATION OF ADA* HIGHER ORDER LANGUAGE TO GUIDANCE AND CONTROL
Published May 1986
114 pages

The need to reduce escalating software life-cycle costs is the raison d'être for Ada. Early experience with the language suggests that the promise of increased software productivity can be fulfulled. However, many problems remain: the need for validated and efficient compilers

* Ada is a registered trademark of the US Government (Ada Joint Program Office)

P.T.O.

AGARD-LS-146

ADA (programming languages)
Computer systems programs
Digital computers
Compilers
Operating systems (computers)

AGARD Lecture Series No.146
Advisory Group for Aerospace Research and Development, NATO
APPLICATION OF ADA* HIGHER ORDER LANGUAGE TO GUIDANCE AND CONTROL
Published May 1986
114 pages

The need to reduce escalating software life-cycle costs is the raison d'être for Ada. Early experience with the language suggests that the promise of increased software productivity can be fulfulled. However, many problems remain: the need for validated and efficient compilers

* Ada is a registered trademark of the US Government (Ada Joint Program Office)

P.T.O.

AGARD-LS-146

ADA (programming languages)
Computer systems programs
Digital computers
Compilers
Operating systems (computers)

AGARD

NATO OTAN

7 RUE ANCELLE · 92200 NEUILLY-SUR-SEINE

FRANCE

Telephone (1) 47.45.08.10 · Telex 610176

DISTRIBUTION OF UNCLASSIFIED AGARD PUBLICATIONS

AGARD does NOT hold stocks of AGARD publications at the above address for general distribution. Initial distribution of AGARD publications is made to AGARD Member Nations through the following National Distribution Centres.Further copies are sometimes available from these Centres, but if not may be purchased in Microfiche or Photocopy form from the Purchase Agencies listed below.

NATIONAL DISTRIBUTION CENTRES

BELGIUM
Coordonnateur AGARD — VSL
Etat-Major de la Force Aérienne
Quartier Reine Elisabeth
Rue d'Evere, 1140 Bruxelles

CANADA
Defence Scientific Information Services
Dept of National Defence
Ottawa, Ontario K1A 0K2

DENMARK
Danish Defence Research Board
Ved Idraetsparken 4
2100 Copenhagen Ø

FRANCE
O.N.E.R.A. (Direction)
29 Avenue de la Division Leclerc
92320 Châtillon

GERMANY
Fachinformationszentrum Energie,
Physik, Mathematik GmbH
Kernforschungszentrum
D-7514 Eggenstein-Leopoldshafen

GREECE
Hellenic Air Force General Staff
Research and Development Directorate
Holargos, Athens

ICELAND
Director of Aviation
c/o Flugrad
Reyjavik

ITALY
Aeronautica Militare
Ufficio del Delegato Nazionale all'AGARD
3 Piazzale Adenauer
00144 Roma/EUR

LUXEMBOURG
See Belgium

NETHERLANDS
Netherlands Delegation to AGARD
National Aerospace Laboratory, NLR
P.O. Box 126
2600 AC Delft

NORWAY
Norwegian Defence Research Establishment
Attn: Biblioteket
P.O. Box 25
N-2007 Kjeller

PORTUGAL
Portuguese National Coordinator to AGARD
Gabinete de Estudos e Programas
CLAFA
Base de Alfragide
Alfragide
2700 Amadora

TURKEY
Department of Research and Development (ARGE)
Ministry of National Defence, Ankara

UNITED KINGDOM
Defence Research Information Centre
Kentigern House
65 Brown Street
Glasgow G2 8EX

UNITED STATES
National Aeronautics and Space Administration (NASA)
Langley Research Center
M/S 180
Hampton, Virginia 23665

THE UNITED STATES NATIONAL DISTRIBUTION CENTRE (NASA) DOES NOT HOLD STOCKS OF AGARD PUBLICATIONS, AND APPLICATIONS FOR COPIES SHOULD BE MADE DIRECT TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS) AT THE ADDRESS BELOW.

PURCHASE AGENCIES

Microfiche or Photocopy
National Technical
Information Service (NTIS)
5285 Port Royal Road
Springfield
Virginia 22161, USA

Microfiche
ESA/Information Retrieval Service
European Space Agency
10, rue Mario Nikis
75015 Paris, France

Microfiche or Photocopy
British Library Lending
Division
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
England

Requests for microfiche or photocopies of AGARD documents should include the AGARD serial number, title, author or editor, and publication date. Requests to NTIS should include the NASA accession report number. Full bibliographical references and abstracts of AGARD publications are given in the following journals:

Scientific and Technical Aerospace Reports (STAR)
published by NASA Scientific and Technical
Information Branch
NASA Headquarters (NIT-40)
Washington D.C. 20546, USA

Government Reports Announcements (GRA)
published by the National Technical
Information Services, Springfield
Virginia 22161, USA

Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

ISBN-92-835-1527-7